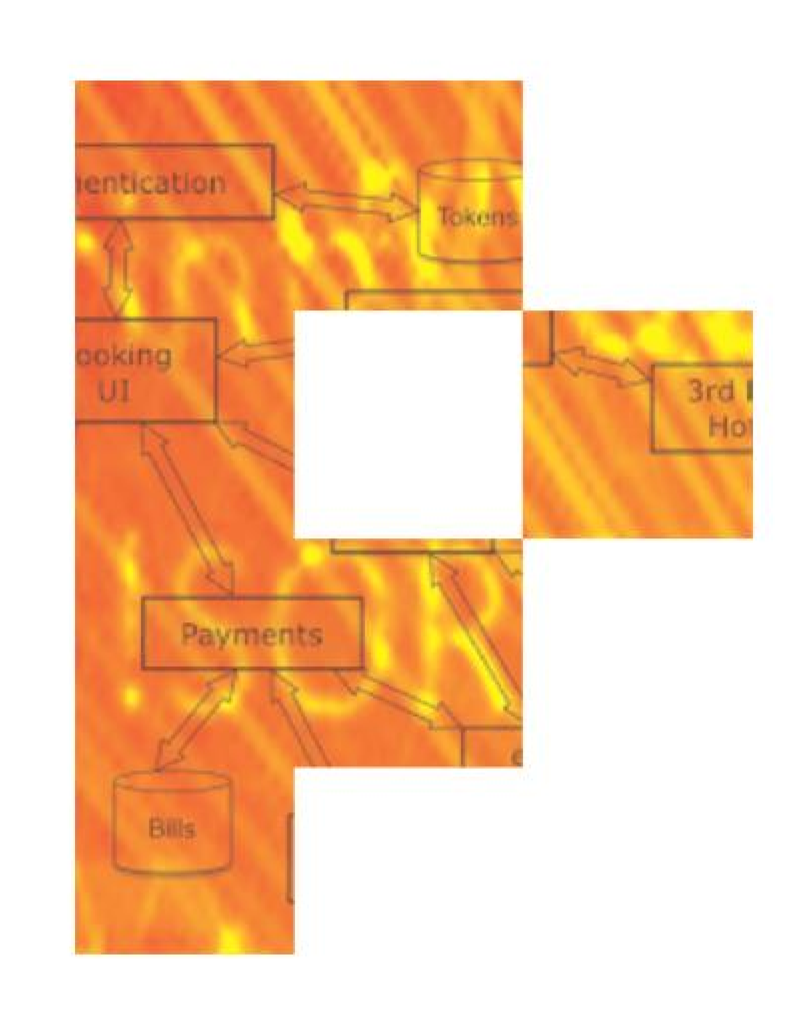
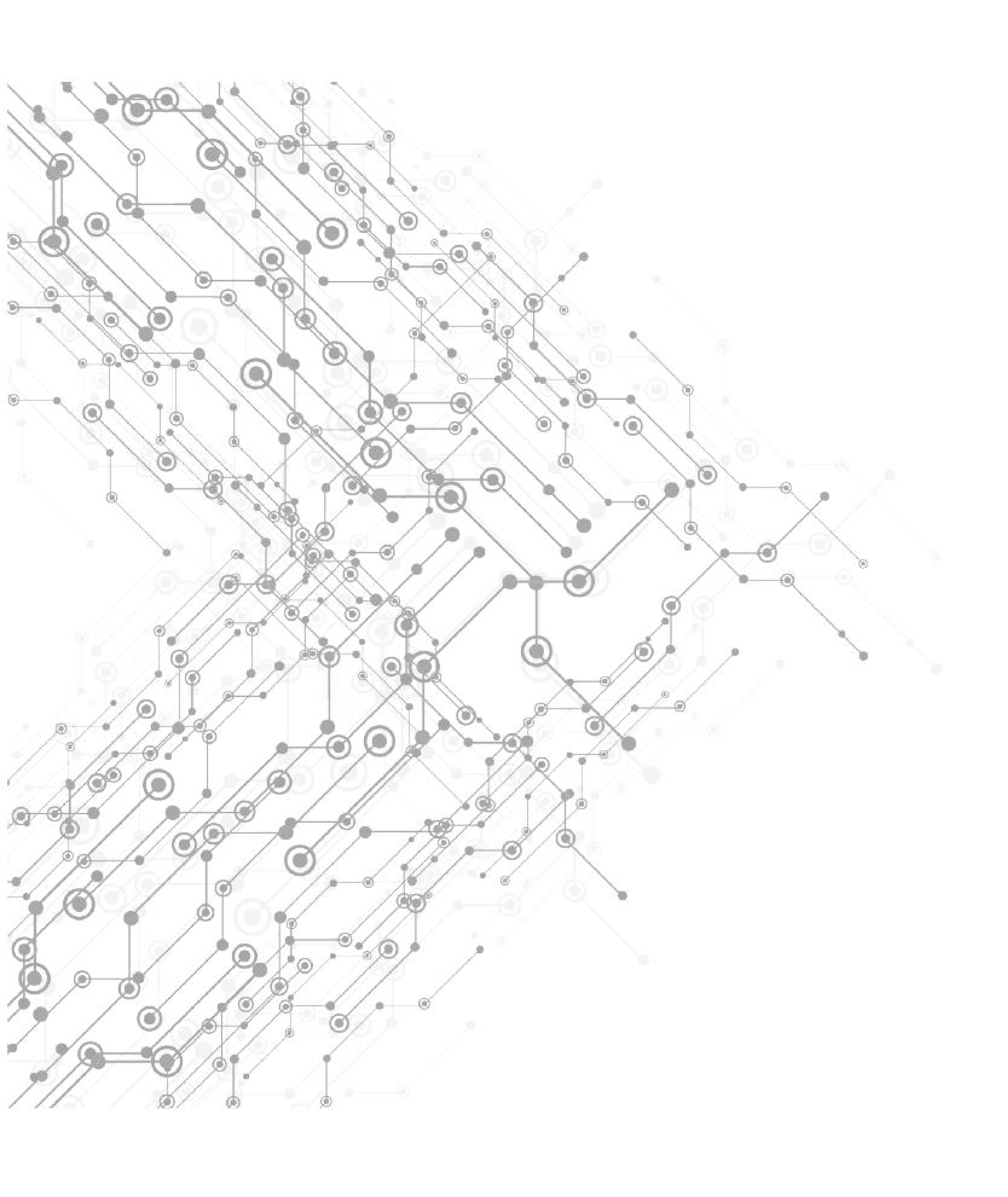
Python Microservices Development



Python
微服务开发

[法] 塔里克·齐亚德（Tarek Ziadé） 著
和坚　张渊 译



清华大学出版社

Python 微服务开发

[法] 塔里克 • 齐亚德(Tarek Ziadé)　著

和坚　张渊　译

清华大学出版社

北　京

图书在版编目(CIP)数据

Python微服务开发 / (法)塔里克 • 齐亚德(Tarek Ziadé) 著；和坚，张渊 译. —北京：清华大学出版社，2019

书名原文：Python Microservices Development
ISBN 978-7-302-52412-0

Ⅰ. ①P…　Ⅱ. ①塔…　②和…　③张…　Ⅲ. ①软件工具—程序设计　Ⅳ. ①TP311.561

中国版本图书馆 CIP 数据核字(2019)第 041781 号

责任编辑：王　军　韩宏志
封面设计：周晓亮
版式设计：孔祥峰
责任校对：牛艳敏
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京鑫丰华彩印有限公司
装 订 者：三河市溧源装订厂
经　　销：全国新华书店
开　　本：170mm×240mm　　**印　　张**：17.75　　**字　　数**：358 千字
版　　次：2019 年 4 月第 1 版　　**印　　次**：2019 年 4 月第 1 次印刷
定　　价：59.00 元

产品编号：080803-01

译者序

近年来，“微服务”技术风靡全球。随着传统互联网和移动互联网的蓬勃发展，企业在快速迭代中积累了大规模服务化开发和运维经验。为通过提高 IT 的响应能力来提升竞争力，微服务架构成为传统企业的“救命稻草”。开发团队在变革中改造和重建技术架构，企业管理者们切实感受到微服务带来的巨大好处。

但不可否认，微服务架构带来了额外复杂性。对开发者提出了更高的要求，开发者不仅要编写业务代码，还需要具有部署和运维等能力。

十多年前，当 Ruby on Rails 和 Django 发布时，人们热衷于追逐包罗万象、开箱即用的全栈式 Web 框架，只需要运行几行脚手架命令，就能快速编写一个包含 Web 页面和数据存储的 Todo List 应用。但时至今日，那些小巧灵便的框架变得越来越受欢迎，开发者更愿意选择“微框架”，通过谨慎地综合运用不同工具来开发应用；随着开发的进行，灵活地升级或替换其中的某些部分。Flask 即是这种微框架之一。

本书模拟真实场景，从一个单体 Flask 应用开始，提出问题，分析和比较方案，作出权衡，最终解决问题(可能引出又一个“问题”——但并非当前的优先级)，逐渐拆分出多个微服务，解决随之而来的部署、监控、安全等新问题，在最后利用异步编程优化性能。期间牵涉大量工具，但本书并未详细介绍它们，只是“点到为止”。本书内容紧贴实用，面向想要开阔眼界和动手实践的开发者和架构师。通过阅读本书，读者将能对微服务开发实践有系统性认识，以便在开发早期进行规划和技术选型。

这里要感谢清华大学出版社的编辑们，他们为本书的翻译投入了巨大热情并付出了很多心血。没有他们的帮助和严谨的要求，本书不可能顺利付梓。

本书涉及大量的实践和专业术语，虽然译者倾力而为，力求译文准确易懂。但毕竟水平有限，失误在所难免，如有任何意见和建议，请不吝指正！本书主要章节由和坚和张渊翻译，参与本书翻译的还有张坤、吴邦、刘易斯、赵汝达、何睿智、刘娟娟、罗冬哲等。

最后，希望读者能通过本书对微服务开发有更深入的理解，并能继续探索解决已知问题的工具和方法。

作 者 简 介

Tarek Ziadé是一位 Python 开发人员，在 Mozilla 的服务团队工作，已使用法语和英语撰写多本 Python 书籍。Tarek 创建了一个名为 Afpy 的法国 Python 用户组，现居住在法国第戎市郊区。在工作之余，Tarek 不忘陪伴家人。他另有两个爱好：跑步和吹小号。

可访问 Tarek 的个人博客(Fetchez le Python)，并在 Twitter 上关注他(@tarek_ziade)。还可在亚马逊上找到他撰写的另一本书 *Expert Python Programming*，该书已由 Packt 出版。

感谢Packt团队，以及帮助过我的以下技术精英：Stéfane Fermigier、William Kahn-Greene、Chris Kolosiwsky、Julien Vehent和Ryan Kelly。

感谢 Amina、Milo、Suki 和 Freya 给予我的爱和耐心支持。

希望在阅读时，你能享受到和我写本书时同样的乐趣！

审校者简介

自 20 世纪 90 年代末以来，William Kahn-Greene 一直在编写 Python 代码和构建 Web 应用。

他在 Mozilla crash ingestion pipeline 的 crash-stats 小组工作，并维护着多种 Python 库，如 bleach。在等待 CI 测试代码改动时，William 会摆弄木制品，照料他种的番茄，并烹饪 4 个人的饭食。

序　　言

7 年前，当我开始在 Mozilla 工作时，为一些 Firefox 功能编写 Web 服务。它们中的一些最终蜕变成微服务。这种变化是随着时间的推移逐渐发生的。促成这种转变的第一个因素是，我们将所有服务转移到云厂商上，并开始与一些第三方服务交互。在云服务上托管应用时，微服务架构成为自然之选。另一个驱动因素是 Firefox 的 Account 项目。我们想在 Firefox 上为用户提供独立身份，以便用户与我们的服务交互。这样一来，所有服务必须与同一个身份提供方(Identity Provider)交互，一些服务器端部分开始重新设计为微服务，以便更高效地工作。

许多 Web 开发者有类似经历，或正在经历这个过程。我也相信 Python 是用来编写小型和高效微服务的最佳语言。Python 生态系统生机勃勃，最新的 Python 3 的特性让它在这个领域中能与过去 5 年中迅猛发展的 Node.js 一决高下。

这就是本书的全部内容。我想分享自己使用 Python 编写微服务的经验，并为此创建了一个简单示例——Runnerly。它位于 GitHub，可供你学习。你可在 GitHub 上与我直接交流，请指出你看到的任何错误，我们可共同切磋如何编写优秀的 Python 应用。

前　言

为将 Web 应用部署到云，代码需要与很多第三方服务进行交互。使用微服务架构，可构建能管理这些交互的大型应用。但这带来一系列挑战，每项挑战都有独特的复杂性。这本通俗易懂的指南旨在帮助你克服这些挑战。书中将介绍如何以最合理的方式设计、开发、测试和部署微服务，紧贴实用的示例将帮助 Python 开发者用最高效的方式创建 Python 微服务。阅读完本书，读者将掌握基于小型标准单元构建大型应用的技能。本书将使用成熟的最佳实践，并分析如何规避常见陷阱。此外，对于正将单体设计转换成新型“微服务”开发范式的社区开发者来说，本书也颇具价值。

本书内容

第 1 章“理解微服务”定义什么是微服务，以及微服务在现代 Web 应用中扮演的角色。还介绍 Python，并解释为什么用 Python 构建微服务是上佳之选。

第 2 章“Flask 框架”介绍 Flask 的主要特性。通过一个 Web 应用示例来展示这个框架，Flask 是构建微服务的基础。

第 3 章“良性循环：编程、测试和写文档”，介绍测试驱动开发方法和持续集成方法，以及在构建和打包 Flask 应用的实践中如何使用这些方法。

第 4 章“设计 Runnerly”基于应用特性和用户案例，首先构建一个单体应用，然后讲述如何将其拆解成微服务，并实现微服务之间的数据交互。还将介绍用来描述 HTTP API 的 Open API 2.0(ex-Swagger)规范。

第 5 章“与其他服务交互”介绍一个服务如何与后台服务进行交互，如何处理网络拆分问题，以及其他交互问题，另外介绍如何独立地测试一个服务。

第 6 章“监控服务”介绍如何在代码中添加日志和指标，清晰地掌控全局，确定发生了什么，并能追查问题和了解服务利用率。

第 7 章“保护服务”介绍如何保护微服务，如何处理用户身份验证、服务间身份验证以及用户管理。还介绍针对服务的欺诈和滥用，以及如何缓解这些问题。

第 8 章“综合运用”描述在终端用户界面中，如何设计和构建一个使用微服务的 JavaScript 应用。

第 9 章“打包和运行 Runnerly”描述如何打包、构建和运行整个应用。开发者必须能够将应用打包到一个开发环境中，确保所有部分都可以运行。

第 10 章“容器化服务”解释什么是虚拟化，如何使用 Docker，如何将服务做成 Docker 镜像。

第 11 章“在 AWS 上部署”首先介绍当前的云服务厂商和 AWS 世界。然后演示如何使用 AWS 来实例化一个基于微服务架构的应用。另外介绍 CoreOS，这是一个专门用于在云上发布 Docker 容器的 Linux 分支。

第 12 章“接下来做什么？”总结全书，在如何构建独立于云厂商和虚拟化技术的微服务问题上，给出一些提示来避免将鸡蛋放入同一个篮子里。还将帮助你巩固第 9 章中学到的知识。

阅读本书需要准备什么

要执行本书的命令和应用，系统需要安装 Python 3.*x*、virtualenv 1.*x* 和 Docker CE。正文中也会根据需要详细列出安装说明。

读者对象

作为一名开发者，如果你了解 Python 基本概念、命令行，以及基于 HTTP 的应用设计原则，并想学习如何构建、测试、扩展和管理 Python 3 微服务，那么本书适合你。阅读本书，你不必具有用 Python 编写微服务的任何经验。

本书约定

代码块按以下样式显示：

```
import time

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
return bytes(json.dumps({'time': time.time()}), 'utf8')
```

会用粗体来显示需要重点关注的代码：

```
from greenlet import greenlet
   def test1(x, y):
   z = gr2.switch(x+y)
print(z)
```

任何命令行的输入或输出都按以下样式显示：

```
docker-compose up
```



警告或重要注释会这样显示。



提示和技巧会这样显示。

读者反馈

欢迎读者提出反馈意见，这样我们能了解你对本书的看法，喜欢什么或不喜欢什么。反馈意见很重要，能帮助我们开发读者真正想了解的主题。只需要发邮件给feedback@packpub.com，并在邮件标题中提及本书，即可将反馈意见发给我们。

如果你是某个主题的专家，有兴趣写书，或愿意为写书做贡献，请到www.packtpub.com/authors 页面查阅作者指南。

下载示例代码

本书相关的代码放在 GitHub 上，网址是 https://github.com/PacktPublishing/Python-Microservices-Development。还有其他代码包和视频，欢迎通过 https://github.com/PacktPublishing/页面下载。

另外，读者可扫描本书封底的二维码直接下载代码。

下载文件后，用以下工具的最新版本来解压缩：

- 在 Windows 系统中使用 WinRAR /7-Zip。
- 在 Mac 系统中使用 Zipeg /iZip /UnRarX。
- 在 Linux 系统中使用 7-Zip /PeaZip。

勘误

尽管我们已经非常小心地确保内容的准确性，但还是会发生失误。如果你在书中发现了错误，可能是文本错误或代码错误，你能向我们报告此事，我们将不胜感激。通过这样做，可减少其他读者的阅读痛苦，并帮助我们改进本书的后续版本。如果你发现任何勘误，请访问 http://www.packtpub.com/submit-errata 页面来报告它们，选择你购买的书，单击 Errata Submission Form 链接，输入勘误的详细信息。一旦填写的勘误被确认，你的提交将被接受，然后勘误将被上传到我们的网站上，或添加到任何现有的勘误列表中。现有的勘误列表位于 Errata 标题的下面。

要查看之前提交的勘误，可访问 https://www.packtpub.com/books/content/support，然后在查找输入框内输入书名。要查找的信息会显示在 Errata 下面。

盗版行为

互联网上的盗版行为是所有媒体一直头疼的问题。在 Packt，我们将尽力处理盗版问题。我们会非常认真地对待版权和许可证的保护。如果你在互联网上遇到任何我们作品的非法拷贝，请立即向我们提供网址或网站名称，以便我们能采取补救措施。

请通过 copyright@packtpub.com 联系我们，并附带上有侵权嫌疑的材料。

非常感激你能帮助保护我们的作者以及我们的工作。这样我们可持续为你带来有价值的内容。

问题

关于本书的任何问题，欢迎通过 questions@packtpub.com 联系。

目　录

第 1 章 理解微服务

软件行业一直在尝试改良软件构建方式。不用说，与穿孔卡片时代相比，当今的软件构建过程已改良了很多。

微服务是过去几年里涌现出的一种改良方法，部分原因是很多公司想缩短发布周期。他们希望能尽快向客户交付新产品或新特性，希望通过迭代达到“敏捷”目的，希望能做到交付、交付、再交付。

如果有大量客户正使用你的服务，相对于发布之前反复测试产品，直接在运行的产品上推送一个试验性功能或移除一项无用的功能是更好的实践方法。

现在，Netflix 等公司正在倡导持续交付技术，这是一种可让小改动频繁上线，然后在一小部分用户中进行测试的技术。他们开发了很多工具，例如 Spinnaker(http://www.spinnaker.io/)就是通过自动执行尽可能多的步骤来更新生产环境，改动的特性通过相互独立的微服务发布到云上。

但如果阅读 Hacker News 或 Reddit，你会发现，梳理出哪些概念真正有用，哪些概念只是随波逐流的新闻体裁是非常困难的。

> “写一篇承诺救赎的论文，使它成为‘结构化’或‘虚拟化’的东西，或使用抽象、分布式、高阶、可适用等概念，你几乎肯定在宣扬一门新邪教。”
>
> ——Edsger W. Dijkstra

本章将讲解什么是微服务，然后重点介绍多个使用 Python 实现微服务的方法。本章要点如下：

- 面向服务架构
- 使用单体方式构建应用
- 使用微服务方式构建应用

- 微服务的益处
- 微服务的缺陷
- 使用 Python 实现微服务

希望当读到本章结尾时，你能深入了解微服务的构建，并明白微服务是什么，以及如何使用 Python。

1.1 SOA 的起源

关于微服务有很多种定义，并没有一个官方标准。在试着解释微服务时，人们通常会提到面向服务架构(Service-Oriented Architecture，SOA)。

> SOA 早于微服务，其核心原则是将应用组织成一个独立的功能单元，可远程访问并单独进行操作和更新。
>
> ——Wikipedia

上述定义中的每个单元都是一个独立服务，它实现业务的一个方面，并通过接口提供功能。

虽然SOA清楚地指出服务应当是独立的进程，但并未强制使用哪种协议进行交互，对如何部署和编排应用还是相当模糊的。

在少数专家于 2009 年发布的 SOA 宣言(http://www.soa-manifesto.org)中，甚至没有提及服务是否通过网络进行交互。

SOA服务可在同一个机器上使用套接字(socket)通过IPC(Inter-Process Communication，进程间通信)方式来交互，如使用共享内存、间接消息队列或远程过程调用(Remote Procedure Call，RPC)。选项非常广泛，只要没有在单个进程中运行所有应用，SOA就可以是任何东西。

常见的说法是，过年几年开始涌现的微服务是 SOA 的一种特定实现方式。它们实现了 SOA 的一些目标，也就是用独立组件来构建应用，组件之间进行着交互。

如果想给出微服务的完整定义，最好先分析一下大多数软件是如何设计架构的。

1.2 单体架构

让我们先通过一个非常简单的例子来介绍传统的单体应用：一个酒店预订网站。

除了静态的 HTML 内容，网站有一个预订功能，可让全球任何城市的用户通过网站预订酒店。用户可搜索酒店，然后用信用卡付款。

当用户搜索酒店网站时，应用将执行以下操作：

(1) 针对酒店数据库执行一些 SQL 查询。

(2) 给合作伙伴的服务发送 HTTP 请求，将更多酒店添加到列表中。

(3) 使用 HTML 模板引擎生成 HTML 结果页面。

一旦用户找到满意的酒店并单击“预订”，应用将执行以下步骤：

(1) 如有必要，在数据库中创建客户，然后进行身份验证。

(2) 通过与银行网络服务交互来完成付款。

(3) 按法律要求，应用需要将支付详情保存到数据库。

(4) 使用 PDF 生成器生成收据。

(5) 用电子邮件服务向用户发送一份用于确认的电子邮件。

(6) 用电子邮件服务将预订电子邮件转发给第三方酒店。

(7) 在数据库中添加用于追踪订单的条目。

上面是一个简化的过程，但紧贴实用。

应用和数据库的交互包括酒店信息、预订信息、支付信息和用户信息等。它还与外部服务进行交互来发送邮件，完成支付，从合作伙伴获取更多酒店。

在经典的 LAMP(Linux-Apache-MySQL-Perl/PHP/Python)架构中，每个传入的请求都会在数据库生成关联的 SQL 查询，以及少量对外部服务的网络请求，然后服务器使用模板引擎生成 HTML 响应。

图 1-1 描述了这种中心化架构。

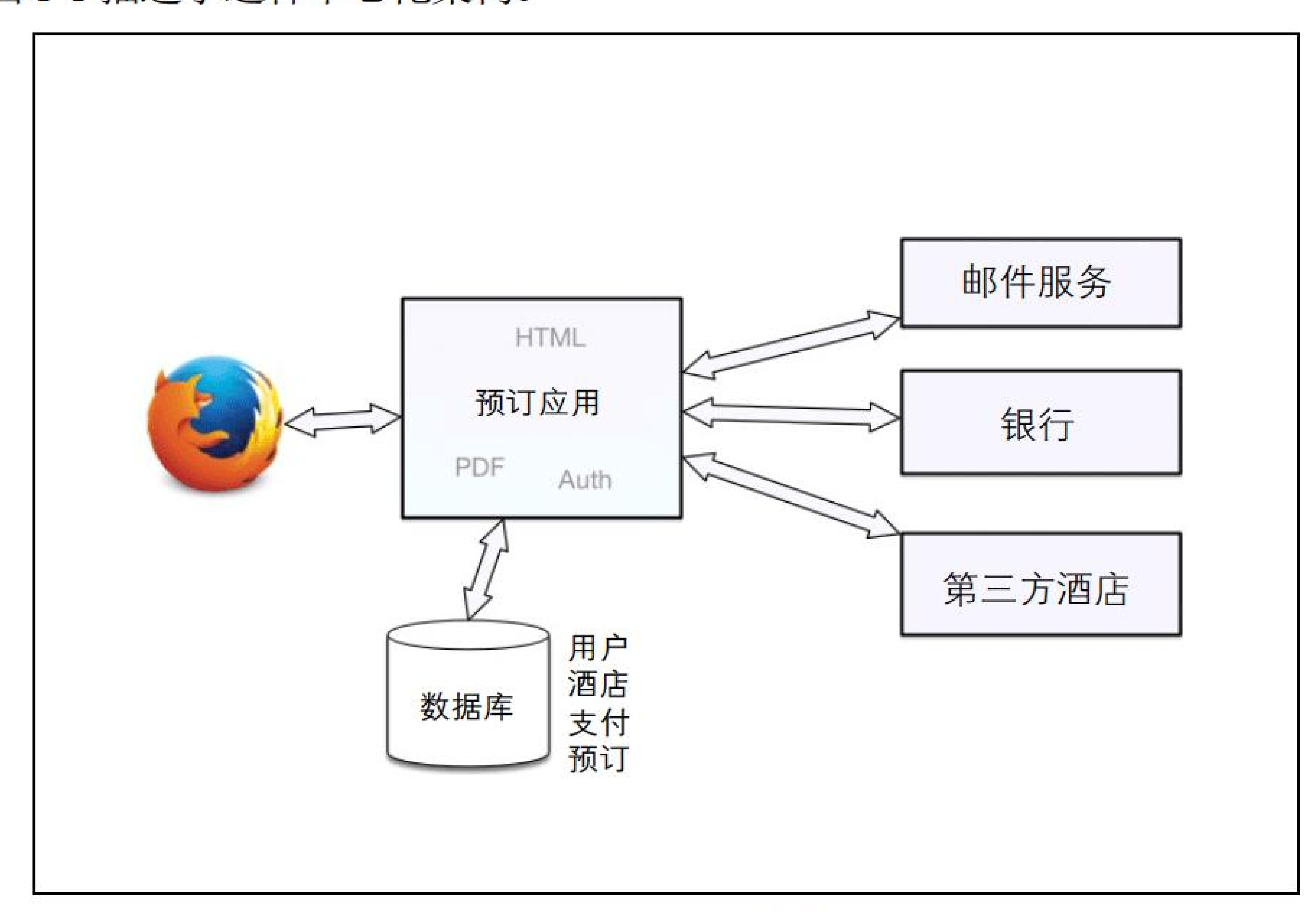


图 1-1　中心化架构

这是一个典型的单体应用，它有很多显而易见的好处。

最大的好处是整个应用程序在一个代码库中，这样开始项目编码就变得十分简单。很容易构建良好的测试覆盖率，还可在一个代码库内以干净和结构化的方式组织代码。将所有数据存储到单一数据库中也简化了应用的开发。可调整数据模型以及代码查询它的方式。

部署也很容易：可给代码库打标签，构建应用包，然后运行它。如果需要扩大规模，可运行多个预订应用的实例，并使用复制机制建立多个数据库。

如果应用一直都很小，这种模型将非常好用，对于单个团队来说，维护也很容易。

但项目通常都会增长，都比最初计划的要大。在一个代码库中维护整个应用会遇到很多棘手的问题。例如，如果要进行一次大范围的彻底修改，如更改银行服务或数据库层，则整个应用会陷入不稳定状态。这些改变在项目生命周期中是个较大的问题，只有通过大量的额外测试才能部署一个新版本。一个项目的生命周期中，这样的改变难免发生。

由于系统的不同部分要求不同的正常运行时间和稳定性，因此一些小变化也会产生附带破坏。例如创建 PDF 出错而导致服务器崩溃，会将付款和预订流程置于风险中，很明显这是存在问题的。

失控性增长是另一个问题，应用迅速添加了很多新特性，不断有开发者离开或加入项目，代码结构变得混乱不堪，测试速度越来越慢。通常，这种增长的最终结果是一个难以维护的意大利面条式的代码库，每次当开发者重构数据模型时，“长毛”的数据库都需要一个复杂的数据迁移计划。

大型软件项目通常需要经历数年时间才能走向成熟，此后，会慢慢地变得难以理解和陷入混乱，最终很难进行维护。这不是因为开发者水平糟糕导致的，而是因为复杂度在增加，很少有人完全理解他们所做的每一个小改动会产生的影响，他们只试图在代码库的某个角落孤立地工作。当从 1 万英尺的高空鸟瞰项目时，看到的只有混乱。

这些都是我们亲身经历过的。

过程是很痛苦的，一个项目开始时，开发者梦想能用最新的架构来构建应用。但紧接着，他们通常会再次陷入同样的困局——熟悉的场景再次上演。

下面总结一下单体应用的优缺点：

- 用单体模式开始一个项目是容易的，可能还是最好的方法。
- 中心化的数据库简化了数据的设计和组织。
- 部署应用较简单。
- 对代码的任何改动会影响原本不相关的功能。对某部分的错误修改可能导致整个应用的崩溃。

- 扩展应用的解决方案存在限制：可部署多个实例，但若其中一个特定功能占用了所有资源，则会影响整个应用。
- 随着代码库的增长，很难保证代码的干净和可控性。

当然也有一些办法可避免上述问题。

常见的解决方案是将应用拆分为不同的部分，而最后生成的代码仍将在单个进程中运行。开发人员通过使用外部库或框架来重构应用从而做到这一点。这些工具可以是内部的，或来自开源软件(Open Source Software，OSS)社区。

如果使用诸如 Flask 的框架在 Python 中构建 Web 应用,可将焦点放在业务逻辑上。最吸引人的地方是能把自己的代码外部化，变成 Flask 的扩展和较小的 Python 包。将代码拆分是控制应用程序增长的好方法。

> “小而美”
>
> ——UNIX 哲学

例如，可使用 Reportlab 和一些模板，将酒店预订应用中的 PDF 生成器拆分成 Python 包。

这个软件包可在其他一些应用中重用，甚至可发布到 Python 包索引(PyPI)中。

但你构建的依然是一个单体应用，很多问题依然存在。例如无法按照不同的部分扩展，缺陷依赖会导致任何间接错误。

还会因为构建时使用了依赖而遇到新挑战。其中一个问题是依赖地狱，如果应用的一部分使用了某个工具库，但 PDF 生成器只能用这个工具库的特定版本，最终将不得不使用一些怪异的解决方案来处理，甚至在分支上定制开发一个修复。

当然本节中描述的所有问题都不可能在项目的第一天出现，而是随着时间推移慢慢堆积起来。

下面看看如果使用微服务来构建相同的应用，会是什么样的。

1.3　微服务架构

如果使用微服务构建相同功能的应用，就可用拆分出的多个组件来管理代码，每个组件运行在独立的线程中。我们不需要使用单一应用负责所有事项，而是如图 1-2 所示拆分成多个微服务。

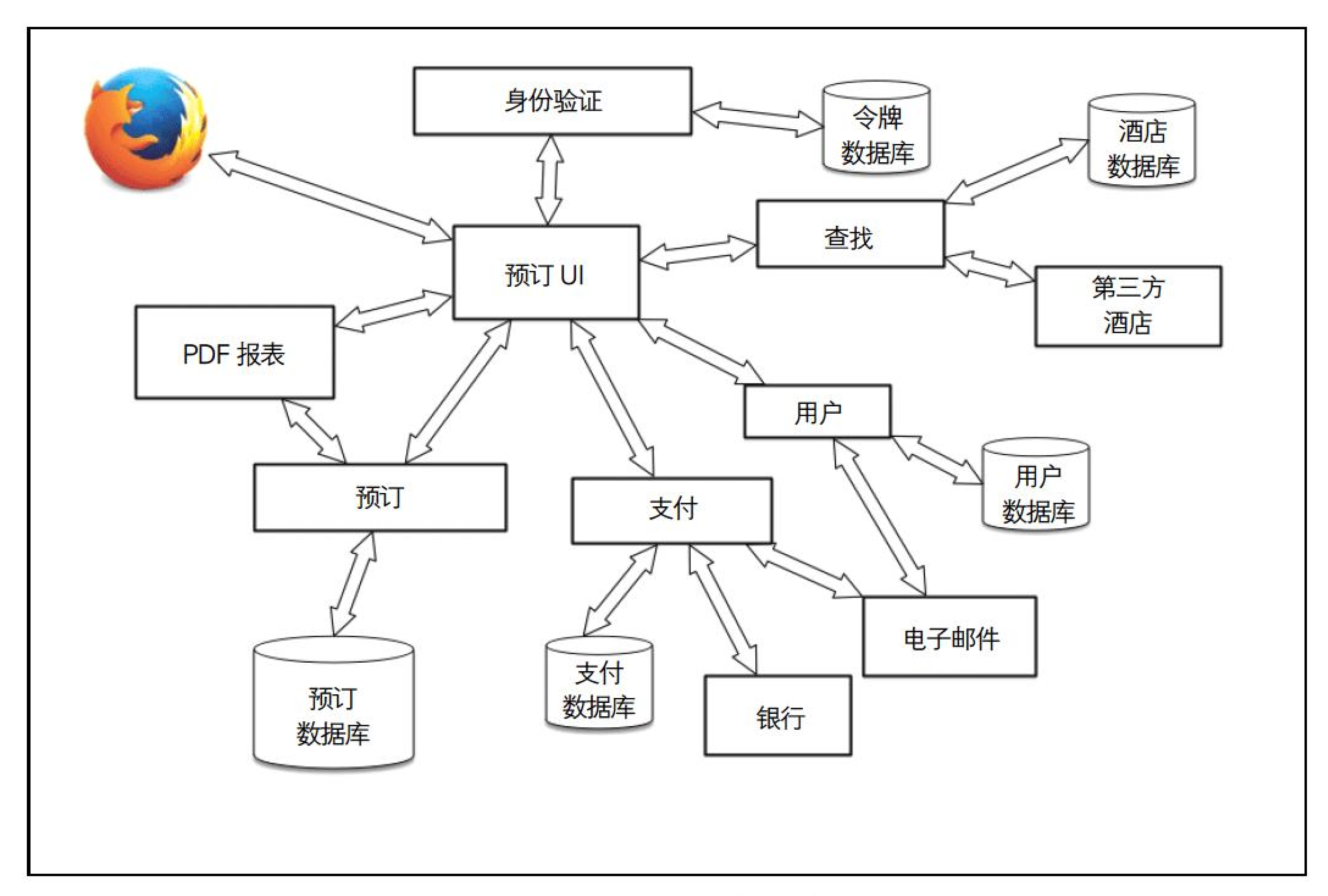


图 1-2　拆分成多个微服务

图中显示的组件数量较多，但不必望而生畏。在单体应用中，内部交互只对内部的某个部分可见。我们已经转移了一些复杂性，最终得到 7 个独立组件：

(1) 预订 UI：一个前端服务，用来生成 Web UI 界面，会与其他所有微服务发生交互。

(2) “PDF 报表”服务：一个非常简单的服务，通过给定的模板和数据把收据或者任何文档创建成 PDF。

(3) 查找：一个可根据城市名查询酒店列表的服务，这个服务有自己的数据库。

(4) 支付：一个和第三方银行服务交互的服务，管理记账数据库。支付成功时会发送电子邮件。

(5) 预订：存储预订信息，并生成 PDF。

(6) 用户：存储用户信息，通过电子邮件和用户交互。

(7) 身份验证：一个基于 OAuth2 来返回身份验证令牌的服务，每个微服务都可在请求其他服务时用它进行身份验证。

这些微服务，连同诸如电子邮件的外部服务，将提供和单体应用相同的功能集。这个架构中的每个组件都使用 HTTP 协议进行通信，通过 REST 风格的 Web 接口提供服务。

由于每个微服务都在内部处理自己的数据结构，所以不需要中心化的数据库，使

用和语言无关的格式(如 JSON)输入和输出数据。可使用任何程序语言都能生成和使用的 XML 或 YAML 格式，最后通过 HTTP 请求和响应进行传输。

“预订UI”服务有些不同，因为它主要用来生成UI页面。依赖于UI使用的前端框架，“预订UI”服务输出的可能是混合的HTML和JSON；如果使用基于静态JavaScript的客户端工具直接在浏览器中生成界面，甚至可以是普通JSON。

除了这个特殊的 UI 情形，使用微服务架构设计的 Web 应用由多个使用 HTTP 进行交互的微服务组成。

这种场景下，微服务是聚焦于特定任务的逻辑单元。这里尝试给出一个完整定义：



微服务是一个轻量级应用，它通过定义良好的契约提供一组有限的功能。它是具有单一责任的组件，可独立开发和部署。

此定义没有提及 HTTP 或 JSON，因为也可以考虑一个基于 UDP 来交换二进制数据的微服务。

但在本书的案例中，所有微服务都是使用 HTTP 协议的简单 Web 应用，都使用和生成 JSON(UI 情形除外)。

1.4　微服务的益处

虽然微服务架构看起来比单体架构复杂得多，但其益处颇多：

- 分离团队的关注点
- 处理更小的项目
- 更多的扩展和部署选项

下面将详细讨论这些内容。

1.4.1　分离团队的关注点

首先，每个微服务可由一个团队独立开发。例如，构建“预订”服务可以是一个完整项目。只要有一个良好的 HTTP API 说明文档，负责开发的团队可使用任何编程语言和数据库。

其次，这意味着应用的演进比单体架构更容易控制。例如，如果支付系统更改其与银行的交互，则影响范围只限于该服务内部，其余应用将保持稳定，甚至完全不受影响。

这种松耦合极大地提高了整个项目的开发速度，从服务层面讲，这是一种类似于

“单一职责原则”的哲学逻辑。

单一职责原则由 Robert Martin 定义，一个类应该只有一个令其改变的原因。换句话说，每个类应该提供单一的、定义良好的功能。在微服务层面，每个微服务都应该聚焦在单个角色上。

1.4.2 更小的项目

第二个益处是降低了项目的复杂度。例如向应用添加一个“PDF 报表”功能时，即便代码很干净，仍会让代码库变得更大，更复杂，有时还会更慢。而在单独应用中构建该功能可避免此问题，因为可更容易地使用任何工具来编程。可频繁地重构它，缩短发布周期，聚焦在最重要的事项上。应用的增长尽在掌控中。

在完善应用时，小项目还能减少风险：如果团队想尝试最新的编程语言或框架，他们可在实现相同微服务 API 的原型上通过快速迭代来试一下，然后决定是否继续使用新的编程语言或框架。

一个浮现在我脑海中的真实例子是 Firefox 的同步存储微服务，通过一些实验，从当前的 Python+MySQL 实现切换到基于 Go 语言的实现，会将用户的数据存储在独立的 SQLite 数据库中。该原型具有很高的实验性质，但由于我们已将存储功能和定义良好的 HTTP API 隔离在一个微服务中，很容易让一小部分用户试用新方案。

1.4.3 扩展和部署

最后，将应用程序拆分为组件更便于在限制下进行扩展。假设每天都有很多顾客预订酒店，而 PDF 生成开始消耗更多 CPU。这时你可将这个特定服务部署到具有更大 CPU 的服务器上。

另一个典型例子是消耗内存的微服务，例如与诸如 Redis 或 Memcache 的内存数据库进行交互。你可通过将微服务部署到具有更少 CPU、更多内存的服务器上来调整部署。

总之，微服务的益处如下：

- 团队可独立开发每个微服务，使用任何技术栈都没问题。可自行定义发布周期。而全部需要定义的只是与语言无关的 HTTP API。
- 开发者将复杂的应用拆分成逻辑单元，每个微服务只关注将自己的事情做好。
- 由于微服务是独立应用，因此可对部署进行更精细的控制，让扩展变得更容易。

微服务架构有利于解决应用开始增长后可能出现的诸多问题。然而，我们需要意

识到，在实践中，微服务架构也会带来一些新问题。

1.5　微服务的缺陷

如前所述，用微服务构建应用有很多益处，但它并非是万能的。

下面列出在编写微服务时，可能需要处理的主要问题：

- 不合理的拆分
- 更多的网络交互
- 数据存储和分享
- 兼容性问题
- 测试

下面将详细讨论这些问题。

1.5.1　不合理的拆分

微服务体系架构的第一个问题是：它如何被设计出来？在第一次尝试中，团队不可能马上想出完美的微服务架构。诸如 PDF 生成器的微服务是个明显的用例。但是，处理业务逻辑时，在你领悟到如何拆分出正确的微服务集之前，你的代码很可能是摇摆不定的。

通过不断试错，设计才能逐渐趋于成熟。而添加或删除微服务可能比重构单体应用更令人痛苦。如果没有证据表明需要拆分，不必先将应用拆分成微服务。

过早拆分是万恶之源。

如果对拆分的意义心存疑问，保持代码在同一个应用中是安全的选择。因为拆分决定可能是错的，所以晚一点把代码拆分到一个新的微服务中比把两个微服务重新合并到一个代码库更容易一些。

例如，如果你总是必须一起部署两个微服务，或一个微服务的改变会影响另一个数据模型，很可能是你没有正确地拆分应用，这两个服务应该重新合并。

1.5.2　更多的网络交互

用微服务构建应用时，第二个问题是会增加网络交互。而在单体版本中，即使代码变得混乱，所有处理也都在一个进程中，可在不需要调用太多后端服务的情况下生成实际响应，然后返回结果。

对于微服务架构，需要额外注意每个后端服务的调用方式，下面是可能出现的问题：

- 如果由于网络隔离或服务延迟，“预订 UI”服务无法调用“PDF 报表”服务，会有什么后果？
- “预订 UI”服务请求其他服务是同步的还是异步的？
- 这将如何影响响应时间？

我们需要有一个坚定的战略来回答所有这些问题，这个主题将在第 5 章中讨论。

1.5.3 数据的存储和分享

另一个问题是数据的存储和分享，有效的微服务需要独立于其他微服务，理想情况下，不应该共享数据库。那么，这对酒店预订应用意味着什么？

再次引发了许多问题：

- 是否在所有数据库中使用相同的用户 ID，或者每个服务都有独立的 ID 并作为隐藏的实现细节来保存？
- 一旦用户添加到系统中，能否通过诸如“数据抽取”的策略将用户信息复制到其他服务数据库中，这么做是不是过度重复了？
- 如何处理数据删除？

以上都是难以回答的问题，本书将介绍许多不同的方法来解决它们。



在设计基于微服务的应用时，一个最大的挑战是如何在保持微服务隔离的同时尽量避免数据重复。

1.5.4 兼容性问题

另一个问题发生在当功能更改影响到多个微服务时。如果不能向后兼容，而且更改影响到服务之间的数据传输方式，你将遇到很多麻烦。

你部署的新服务能否与旧版本的其他服务一起使用？或者你是否需要一次修改和部署多个服务？这是不是说你可能无意中发现一些应该合并在一起的服务？

良好的版本控制和干净的 API 设计有助于缓解这些问题，本书后面将详细讲解这个问题。

1.5.5 测试

最后，如果要进行端到端的测试并部署整个应用，现在需要测试很多积木一样的

微服务。要有一个强健且敏捷的过程才能高效部署。在开发时要顾及完整应用。你不可能做到只根据其中一个微服务就完整地测试整个应用。

幸运的是，现在有许多工具可帮助部署使用多个组件构建的应用，我们也将在书中学到这些工具。所有这些工具推动了微服务架构的成功和采用；如果没有这些工具，微服务也不会是今天的面貌。



微服务风格架构促进了部署工具的革新，部署工具降低了微服务风格架构的获准门槛。

下面总结一下微服务的缺陷：

- 过早将应用拆分成微服务可能导致架构设计问题。
- 微服务之间的网络交互增加了开销。
- 测试和部署微服务较为麻烦。
- 最大的挑战：不同微服务之间很难共享数据。

本节提到的所有这些缺陷其实都不必过于担心。它们看起来似乎难以应对，传统的单体应用好像更安全一些。但从长远看，通过将项目拆分成微服务，开发和运维工作都变得更容易了。

1.6　使用 Python 实现微服务

Python 是一门神奇的多用途语言。

你可能已经知道，Python 可用来构建很多不同类型的应用程序，从用来执行服务器任务的简单系统脚本，到为数百万用户提供服务的大型面向对象应用。

根据 Philip Guo 在 2014 年发布在美国计算机协会(Association for Computing Machinery)网站上的一项研究，Python 在美国顶尖大学的使用率已经超过 Java，成为学习计算机科学最流行的语言。

这一趋势在软件行业也是如此。Python 现在位列 TIOBE 索引(http://www.tiobe.com/tiobe-index/)的前五名，在 Web 开发领域的市场份额可能更大，因为像 C 这样的语言很少被用来构建 Web 应用程序。



本书假设你已经熟悉 Python 编程语言。如果你还不是一个富有经验的 Python 开发者，可阅读本书作者的另一本书 *Expert Python Programming, Second Edition*，来学习高阶 Python 编程技能。

有些开发者批评 Python 的速度慢，不适合构建 Web 服务。Python 的确有些慢，

但依然是构建微服务的语言选项，许多大公司都乐意使用它。

本节将给出使用不同方法来构建 Python 微服务的背景，还给出关于异步与同步编程的一些深刻见解，最后总结有关 Python 性能的一些细节。

本节包含 5 个部分：

- WSGI 标准
- greenlet 和 gevent 模块
- Twisted 和 Tornado 模块
- asyncio 模块
- 语言性能

1.6.1 WSGI 标准

可以很方便地用 Python 创建和运行 Web 应用，这是 Python 吸引大多数 Web 开发者的原因。

受到公共网关接口(Common Gateway Interface，CGI)的启发，Python Web 社区建立了一个标准，称为 WSGI (Web Server Gateway Interface，Web 服务器网关接口)。有了 WSGI，可更方便地编写 Python 应用来支持 HTTP 请求。

使用这个标准编码时，即可通过 uwsgi 或 mod_wsgi 等 WSGI 扩展，由 Apache 或 nginx 等标准服务器执行项目。

应用只需要处理传入请求并返回 JSON 响应，Python 在标准库中包含了所有实现细节。

使用普通 Python 模块，只需要不到 10 行代码，即可创建一个返回服务器本地时间的功能完备的微服务。下面是代码：

```
import json
import time

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    return [bytes(json.dumps({'time': time.time()}), 'utf8')]
```

自从 WSGI 协议建立，它就成为一个重要标准，Python Web 社区广泛采用了它。开发者通过编写可挂在 WSGI 应用之前或之后的功能性中间件，在 Web 应用环境中完成不同的事情。

一些 Web 框架，如 Bottle(http://bottlepy.org)，就是专门根据该标准创建的。此后，

很多框架都可在 WSGI 协议下使用。

使用 WSGI 的最大问题是原生同步性。对于每个传入请求，都会调用上述代码中的 application 函数一次，当函数返回时，必须发回响应。这意味着，每次调用 application 函数时，在响应准备好之前，都会阻塞。

对于这种情况下编写的微服务，代码总是需要等待各种网络资源的响应。换句话说，这时你的应用是停顿的，在所有东西准备好之前，客户端会被阻塞。

对 HTTP API 来说，这样做问题不大。我们并非讨论构建双向应用(如基于 Web 套接字的应用)。但是，当你的应用同时收到多个调用请求时会发生什么？

WSGI 服务器允许运行一个线程池，来并发服务多个请求。但你不可能运行几千个线程；一旦线程池耗尽，即便微服务除等待后端服务响应外无所事事，下一个请求还是会阻塞客户的访问。

出于上述原因及其他因素，诸如 Twisted 或 Tornado 的非 WSGI 框架，以及 JavaScript 领域的 Node.js 都大获成功，因为它们完全是异步框架。

在编写 Twisted 应用时，可使用回调来暂停和恢复生成响应的工作。此时，可接受一个新请求并开始处理。该模型极大地缩短了进程的停顿时间。可服务数千个并发请求。当然，这并非说应用会更快地返回每个响应，只是说一个进程可接受更多并发请求，在数据准备好之前，能在请求间进行切换。

在 WSGI 标准中没有一个简单方式可做到同样的事情，虽然社区内争论了多年，但最终没能达成共识。社区最终可能放弃 WSGI 标准。

同时，如果你的部署考虑到 WSGI 标准的“一个请求对应一个线程”限制，那么使用同步框架构建微服务仍然是可能的。

不过还有一个诀窍来提升同步的 Web 应用，这就是 greenlet，下一节将对此进行解释。

1.6.2　greenlet 和 gevent 模块

异步编程的一般原则是，让进程处理多个并发执行的上下文来模拟并行处理方式。

异步应用使用一个事件循环，当一个事件触发时暂停或恢复执行上下文；只有一个上下文处于活动状态，上下文之间进行轮替。代码中的显式指令将告诉事件循环，哪里可暂停执行。这时，进程将查找其他待处理的线程进行恢复。最终，进程将回到函数暂停的地方并继续运行。从一个执行上下文移到另一个称为“切换”。

greenlet 项目(https://github.com/python-greenlet/greenlet)是根据 Stackless 项目构建的程序包，是一个特别的 CPython 实现。

greenlet 是易于实例化的伪线程，可用来调用 Python 函数。在这些函数中，可切

换到另一个函数，即将控制权交给另一个函数。切换是通过事件循环完成的，允许使用类似线程的接口范式来编写异步应用。

下面是 greenlet 文档中的一个例子：

```
from greenlet import greenlet
def test1(x, y):
    z = gr2.switch(x+y)
    print(z)

def test2(u):
    print (u)
    gr1.switch(42)

gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch("hello", " world")
```

上例中的两个 greenlet 显式地从一个切换到另一个。

为构建基于 WSGI 标准的微服务，如果底层代码使用 greenlet，我们可接受多个并发请求，当知道一个调用将阻塞请求(如 I/O 请求)时，只需要从一个请求切换到另一个。

不过，从一个 greenlet 切换到另一个需要显式地完成，这会导致代码变得混乱，难以理解。这就轮到 gevent 大显身手了。

gevent(http://www.gevent.org/)项目构建在greenlet的上层，能采用隐性方式在greenlet之间自动切换，它还有其他许多功能。

gevent 提供了 socket 模块的协作版本，当 socket 中的一些数据准备好后，会使用 greenlet 自动地暂停或恢复执行。甚至有一个 monkey patch 功能，可自动用 gevent 版本的 socket 来替代标准库 socket。只需要通过一行额外代码就可让你的标准同步代码魔术般地每次都异步使用 socket：

```
from gevent import monkey; monkey.patch_all()

def application(environ, start_response):
    headers = [('Content-type', 'application/json')]
    start_response('200 OK', headers)
    # ...do something with sockets here...
    return result
```

当然这种隐含的魔力是有代价的。为使 gevent 正常工作，所有底层代码都需要与 gevent 补丁兼容。来自社区的一些包可能因此继续阻塞或返回非期望的结果；如果这些包使用 C 语言扩展，并绕过了打上 gevent 补丁的标准库的一些功能，这表现得尤其明显。

不过大多数情况下都能正常工作。而且，与 gevent 配合良好的项目会被标记成绿色，如果一个库不能与 gevent 配合，社区通常会要求库的作者修复成绿色。

例如，在 Mozilla，这用来扩展 Firefox Sync 服务。

1.6.3　Twisted 和 Tornado 模块

如果增加并发请求数量对你构建的微服务很重要，可尝试放弃WSGI标准，而使用诸如Tornado(http://www.tornadoweb.org/)或Twisted(https://twistedmatrix.com/trac/)的异步框架。

Twisted 已经存在多年。要实现相同的微服务，需要编写的代码要略微长一些，如下所示：

```
import time
import json
from twisted.web import server, resource
from twisted.internet import reactor, endpoints

class Simple(resource.Resource):
    isLeaf = True
    def render_GET(self, request):
    request.responseHeaders.addRawHeader( b"content-type",
                                          b"application/json")
    return bytes(json.dumps({'time': time.time()}), 'utf8')

site = server.Site(Simple())
endpoint = endpoints.TCP4ServerEndpoint(reactor, 8080)
endpoint.listen(site)
reactor.run()
```

虽然 Twisted 是一个非常健壮和高效的框架，但使用它构建 HTTP 微服务时，你可能遇到下列问题：

- 必须使用从 Resource 类派生的类(该类实现了每个支持的方法)来实现微服务的每个端点。对于简单的 API 来说，它增加了很多样板代码。

- 由于是原生异步的，Twisted 代码可能很难理解和调试。
- 当你串联了很多功能，又将其逐一触发时，很容易掉进回调地狱——代码会变得混乱。
- 很难正确测试 Twisted 应用，你必须使用 Twisted 特定的单元测试模型。

Tornado基于类似的模型，但在某些领域做得更好。它有一个量级更轻的路由系统，并尽可能使代码更接近普通Python。Tornado也使用回调模型，所以调试难度较大。

但这两个框架都努力借助 Python 3 中引入的新异步功能来弥补差距。

1.6.4 asyncio 模块

当 Guido van Rossum 开始在 Python 3 中添加异步功能时，社区的一部分人认为，以同步、顺序化方式编写应用更合理一些，不必像 Tornado 或 Twisted 那样必须添加显式回调。这些人推荐使用类似 gevent 的解决方案。

但 Guido 选择了显式技术，并在 Tulip 项目(该项目受到 Twisted 的启发)进行尝试。最后，asyncio 模块从 Tulip 项目中诞生，并添加到 Python 中。

事后看来，在 Python 中实现显式的事件循环机制，而非采用 gevent 的方式是比较合理的。Python 核心开发人员编写了 asyncio，优雅地使用 async 和 await 关键字来扩展 Python 语言实现协程(coroutine)，使用普通 Python 3.5+构建的异步应用代码看起来非常优雅，而且很接近同步编程。



协程是能暂停和恢复程序执行的功能。第 12 章将详细解释如何在 Python 中实现协程以及如何使用协程。

回调语法混乱问题经常出现在 Node.js 和 Twisted(Python 2)应用中。但通过以上方式，Python 很好地避免了此类问题。

除了协程外，Python 3 还在 asyncio 包中引入一套完整的功能和帮助程序来构建异步应用，详情可参阅 https://docs.python.org/3/library/asyncio.html。

Python 现在可像 Lua 这样的表达式语言一样创建基于协程的应用，现在有一些新框架已嵌入这些功能。只有 Python 3.5+版本支持此功能。

KeepSafe 的 aiohttp(http://aiohttp.readthedocs.io)就是其中之一，只需要几行优雅的代码就能构建完全异步的微服务：

```
from aiohttp import web
import time


async def handle(request):
```

```
    return web.json_response({'time': time.time()})

if __name__ == '__main__':
    app = web.Application()
    app.router.add_get('/', handle)
    web.run_app(app)
```

在这个简短示例中，实现方式非常类似于同步应用的实现方式。使用异步的唯一提示是 async 关键字，async 关键字用来指出 handle 函数是协程的。

这就是将在异步 Python 应用的每个级别上使用的方式。这里有另一个使用 aiopg 的例子，来自 asyncio 的 PostgreSQL 库的项目文档：

```
import asyncio
import aiopg

dsn = 'dbname=aiopg user=aiopg password=passwd host=127.0.0.1'

async def go():
    pool = await aiopg.create_pool(dsn)
    async with pool.acquire() as conn:
        async with conn.cursor() as cur:
            await cur.execute("SELECT 1")
            ret = []
            async for row in cur:
                ret.append(row)
            assert ret == [(1,)]

loop = asyncio.get_event_loop()
loop.run_until_complete(go())
```

通过添加少量 async 和 await 前缀，让执行 SQL 查询和返回结果的函数看起来接近于同步函数。

但基于Python 3的异步框架和库还处于兴起阶段，如果你使用asyncio或诸如aiohttp的框架，都必须在每个需要的功能中使用特定的异步实现方式。

如果代码中需要使用非异步的库，那么需要完成一些额外的和富有挑战性的工作，以免阻塞事件循环。

如果你的微服务处理的资源数量有限，则是可管理的。但在撰写本书期间，坚持使用已经成熟的同步框架可能比使用异步框架更安全。让我们先享用目前成熟的程序包的生态系统，并期盼 asyncio 生态系统早日走向成熟！

Python 中有很多同步框架可用来构建微服务，如 Bottle、Pyramid with Cornice、Flask 等。



本书的下一个版本很可能使用异步框架。但这一版中还是使用 Flask 框架。Flask 已经存在了一段时间，非常健壮和成熟。但请记住，无论使用什么 Python Web 框架，都应能接替运行本书中的所有示例。这是因为所有构建微服务的代码都非常接近于普通 Python，而框架的主要作用是路由请求并提供一些帮助。

1.6.5 语言性能

前一节探讨了两种编写微服务的不同方式：异步和同步。无论使用哪种技术，Python 的速度直接影响着微服务性能。

当然，每个人都知道 Python 比 Java 和 Go 要慢，但执行速度并非总是最重要的。微服务通常是一层薄薄的代码，它的大部分时间都在等待来自其他服务的网络响应。Postgres 服务器需要快速返回 SQL 查询结果(因为构建响应时，其中花费的时间最多)，与此相比，对于微服务而言，内核速度就没那么重要了。

当然，让应用尽快运行是合情合理的要求。

在Python社区中，围绕语言加速的一个有争议的话题是GIL(Global Interpreter Lock)互斥会破坏性能，因为多线程应用不能使用多个进程。

GIL 的存在是有理由的，它可保护 CPython 解释器的非线程安全部分，也存在于 Ruby 等其他语言中。到目前为止，所有试图将其删除的尝试都未能加快 CPython 实现的速度。



Larry Hasting 正在研究一个名为 Gilectomy(https://github.com/larryhastings/gilectomy)的无GIL CPython项目。它的最低目标是提出一个无GIL实现，能使单线程应用的运行速度达到CPython级别。但到撰写本书时为止，还是慢于CPython的。跟踪这个项目，看它能否达到“速度相同”的那一天很有趣，那时，非GIL CPython将非常有吸引力。

对于微服务，除了阻止在同一进程中使用多个内核外，GIL 也会由于互斥锁带来的系统调用开销而降低高负载时的系统性能。

然而，围绕 GIL 的所有关注是有益的，在过去几年，已经做了一些工作来减少解释器中的 GIL 争夺，这让 Python 的性能在一些领域有了极大提高。

注意，即使核心团队删除 GIL，由于 Python 是一门解释性和垃圾收集语言，也会遭受这些特性带来的性能损失。

如果你对解释器如何分解函数感兴趣，可分析一下 Python 提供的 dis 模块。下面的示例中，解释器将一个生成自增值的函数分解成不少于 29 步的指令操作序列：

```
>>> def myfunc(data):
...     for value in data:
...         yield value + 1
...
>>> import dis
>>> dis.dis(myfunc)
  2            0 SETUP_LOOP            23 (to 26)
               3 LOAD_FAST              0 (data)
               6 GET_ITER
        >>     7 FOR_ITER              15 (to 25)
              10 STORE_FAST             1 (value)

  3           13 LOAD_FAST              1 (value)
              16 LOAD_CONST             1 (1)
              19 BINARY_ADD
              20 YIELD_VALUE
              21 POP_TOP
              22 JUMP_ABSOLUTE     7
        >>    25 POP_BLOCK
        >>    26 LOAD_CONST             0 (None)
              29 RETURN_VALUE
```

用静态编译的语言编写的类似函数将大大减少生成相同结果所需的操作数。不过，还有一些方法可加快 Python 的执行速度。

一种方法是构建 C 语言扩展，或使用诸如 Cython(http://cython.org/)的语言静态扩展，将代码的一部分写入编译代码，但这会使代码更复杂。

另一种最有前景的解决方案是，仅使用 PyPy 解释器(http://pypy.org/)来运行应用。

PyPy 实现了 JIT(Just-In-Time)编译器，此编译器在运行时直接用 CPU 可使用的机器码替换 Python 片段。JIT 的策略是在执行前，实时检测何时编译以及如何编译。

虽然 PyPy 总比 CPython 滞后几个 Python 版本，但你可在生产环境中使用它，而且它的性能惊人。我们有一个 Mozilla 项目需要快速执行，PyPy 版本的程序几乎与 Go 版本的程序一样快，因此我们在那里改用 Python。



要了解 PyPy 与 CPython 的不同之处，PyPy Speed Center 网站(http://speed.pypy.org/)是可供访问的绝佳场所。

如果你的程序使用了 C 扩展，则需要针对 PyPy 重新编译，这是一个问题。如果其他开发人员维护你使用的某些扩展，这尤其麻烦。

不过，如果你用一组标准库构建微服务，则很可能可直接与 PyPy 解释器一起工作，所以这是值得一试的。

对于大多数项目而言，Python 及其生态系统的好处大大超出本节描述的性能问题，因为微服务的性能开销很少是一个问题。即使算作一个问题，微服务方法也允许你在不影响系统其余部分的情况下，重新编写“性能关键型”组件。

1.7　本章小结

本章比较了构建 Web 应用所用的单体方法和微服务方法。很明显，这两个选项并非严重对立，你并不需要在第一天就做出抉择并坚持到底。

可使用单体来启动项目，然后用微服务加以改进。随着项目逐渐成熟，部分服务逻辑应迁移到微服务中。这是你从本章学到的有用方法，但需要小心地避免落入一些常见陷阱。

另一个要点是，Python 被认为是编写 Web 应用的最佳语言之一，也是编写微服务的最佳语言之一。由于 Python 提供了大量成熟的框架和包，在其他领域也被经常使用。

本章简单介绍了几个同步或异步框架，在本书的后续章节，将使用 Flask 框架。

下一章将介绍奇妙的 Flask 框架，即使你之前不熟悉它，也很可能会喜欢上它。

最后，Python 是一个较慢的语言，在某些特定的情况下这可能是一个问题。但通过弄清楚是什么让它变慢，几个避免缓慢的解决方案足以让这个问题变得不再重要。

第 2 章 Flask 框架

Flask 出现于 2010 年左右，它利用了 Werkzeug WSGI 工具包(http://werkzeug.pocoo.org/)和其他多种工具(如路由系统)；其中 Werkzeug WSGI 工具包为通过 WSGI 协议与 HTTP 请求交互奠定了基础。

Werkzeug 在功能层面相当于 Paste。Pylons 项目(http://pylonsproject.org)是一个伞状组织(包含另一个 Web 框架 Pyramid 项目)，在一定程度上集成了 Paste 和许多组件。

上面这些项目与 Bottle(http://bottlepy.org/)以及其他一系列项目一起，组成了 Python 微框架生态系统。

所有这些项目有一个共同目标：为 Python 社区提供简单工具，帮助应用开发者更快地构建 Web 应用。

不过，术语“微框架(microframework)”可能令人产生一些误解。它并不意味着只能构建微小的应用。通过这些工具，将能构建任何应用——甚至一个大型项目。“微”前缀的含义是：这些框架尽量少做技术决策，而将决策权交给应用开发者。你可按自己的方式组织应用代码，并使用自己喜欢的任何库。



微框架充当“胶水代码”的角色，它将请求传递到系统，然后返回响应，它不会强迫你的项目使用某种特定模式。

这种哲学理念的一个典型例子发生在当需要与 SQL 数据交互时。例如，诸如 Django 的“内置电池”式框架提供了构建 Web 应用需要的所有组件，包括用来绑定对象和数据库查询结果的对象关系映射框架(Object-Relational Mapper，ORM)。这个框架的其余部分与 ORM 紧密集成。

如果想在 Django 中使用其他 ORM 框架(如 SQLAlchemy)，以利用那些框架的某些强大功能，却并不容易。因为 Django 的整体理念是提供一个完整的工作系统，这样

开发人员能专注于构建最初的业务功能。

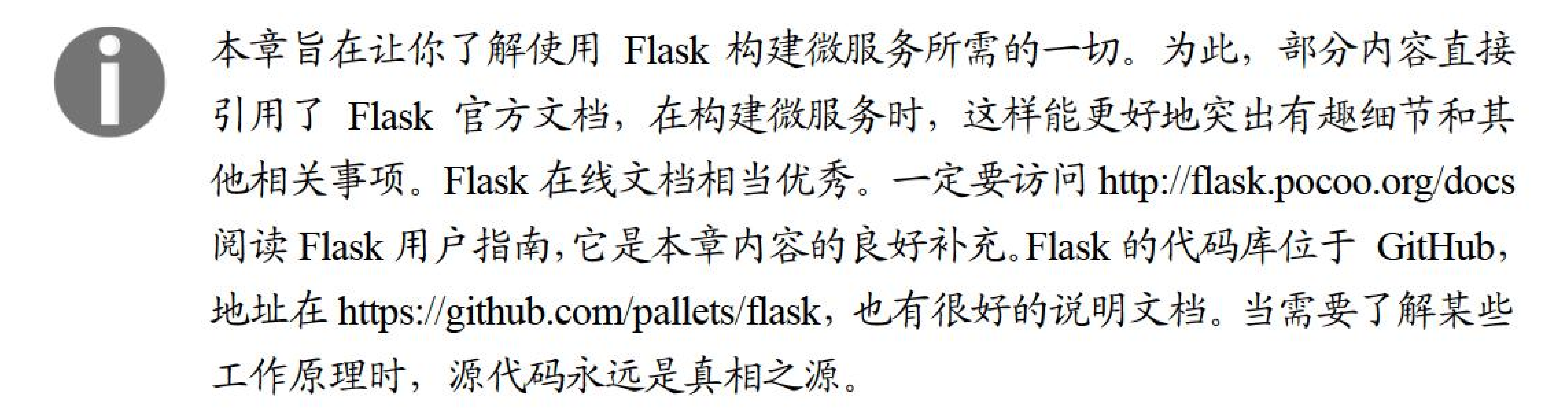
作为另一套生态，Flask框架并不关心与数据交互的库。Flask 仅试图确保它有充足的钩子，以便让外部库能通过扩展来实现各种功能。即，如果想在Flask中使用SQLAlchemy，并正确地使用SQL会话和事务，很可能需要在项目中添加诸如Flask-SQLAlchemy的软件包。如果你不喜欢这个软件包集成 SQLAlchemy的方式，那么可以选择另一个，甚至可自行开发这部分集成工作。

当然，这种方式不是完美方案。完全自由地选择，意味着很容易作出糟糕决定，导致构建出的应用存在问题，要么依赖于有缺陷的库，要么设计不当。

先不必担心这些问题！本章将确保你理解 Flask 能提供什么，以及在构建微服务时如何组织代码。

本章涵盖如下主题：

- 使用哪个版本的 Python
- Flask 如何处理请求
- Flask 的内置特性
- 一个微服务骨架


本章旨在让你了解使用 Flask 构建微服务所需的一切。为此，部分内容直接引用了 Flask 官方文档，在构建微服务时，这样能更好地突出有趣细节和其他相关事项。Flask 在线文档相当优秀。一定要访问 http://flask.pocoo.org/docs 阅读 Flask 用户指南，它是本章内容的良好补充。Flask 的代码库位于 GitHub，地址在 https://github.com/pallets/flask，也有很好的说明文档。当需要了解某些工作原理时，源代码永远是真相之源。

2.1 选择 Python 版本

在深入介绍 Flask 前，先回答一个问题。Flask 能支持多个版本的 Python，那么应该使用哪个版本？

前一章提到过，Python 3 已经有了令人难以置信的进步。与 Python 3 不兼容的软件包已经很罕见了。除非要构建非常特殊的应用，否则直接使用 Python 3 没有任何问题。

基于微服务的架构意味着每个应用运行在独立环境中。所以，根据项目的具体情况，让某些应用使用 Python 2，让另一些应用使用 Python 3 也是可行的。你甚至可使用 PyPy。

针对是否采用 Python 3 这个问题，Flask 的创立者在早期并没有给出明确回复。但现在的 Flask 文档明确指出，新项目应该开始使用 Python 3。参见 http://flask.pocoo.org/docs/latest/python3/#python3-support。

由于 Flask 并没有使用 Python 3 的任何最前沿语言特性，因此，应用的代码最终可能在 Python 2 和 Python 3 上都可运行。在最糟的情况下，根据需要可使用诸如 Six 的工具，让代码能同时支持两个版本。

如果不是因为限制条件必须使用 Python 2，那么通常建议使用 Python 3。2020 年后，Python 2 将不再受支持。参见 https://pythonclock.org/。



本书采用最新的Python 3.5 稳定发布版编写所有示例代码，它们在最新的 Python 3.x上也应该能正常运行。现在，请确保已经准备好Python 3 环境，并安装了virtualenv(https://virtualenv.pypa.io)。本书的所有示例代码都运行在终端上。

2.2 Flask 如何处理请求

Flask 框架的入口是 flask.app 模块里的 Flask 类。运行一个 Flask 应用时，其实是运行这个类的单一实例，它负责处理传入的 WSGI 请求，然后分发给正确的处理代码，最终返回响应。



WSGI 规范定义了 Web 服务器和 Python 应用之间的接口。这个规范使用单一映射来描述传入请求，此后，诸如 Flask 的框架负责将请求路由给正确的可调用对象。

Flask 类提供了路由(route)方法，该方法可装饰函数。当一个函数被装饰后，就会成为一个视图(view)，并被注册到 Werkzeug 的路由系统中。这个系统使用一个极小的规则引擎来匹配传入请求和视图；稍后将详细介绍这个过程。

下面是一个简单但完整的 Flask 应用：

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.route('/api')
def my_microservice():
    return jsonify({'Hello': 'World!'})
```

```
if __name__ == '__main__':
    app.run()
```

访问/api 时，应用会返回一个 JSON 映射。如果访问其他任何路径，应用会返回 404 错误。

变量__name__是这个应用软件包的名称，当运行一个单独的 Python 模块时，变量__name__会赋值为__main__。Flask 用这个变量实例化一个新的日志记录器(logger)，并在磁盘上定位这个模块所在文件的路径。Flask 将使用该文件的目录作为助手程序(例如与应用程序相关的配置文件)的根目录，并根据此目录确定静态文件目录(static)和模板目录(templates)的默认存放位置。

在 shell 中运行这个模块时，Flask 会运行其内置的 Web 服务器，并在 5000 端口上监听传入的请求。

```
$ python flask_basic.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

当使用 curl 命令访问/api 时，会得到一个合法的 JSON 响应和正确的消息头。这多亏了 jsonify()函数，该函数负责将 Python 字典类型转换成合法的 JSON 响应，并添加适当的 Content-Type 消息头。



本书大量使用 curl 命令。Linux 或 macOS 已经预装了这个命令。请参见 https://curl.haxx.se/。

```
$ curl -v http://127.0.0.1:5000/api
* Trying 127.0.0.1...
...
< HTTP/1.0 200 OK
< Content-Type: application/json
< Content-Length: 24
< Server: Werkzeug/0.11.11 Python/3.5.2
< Date: Thu, 22 Dec 2016 13:54:41 GMT
<
{
    "Hello": "World!"
}
```

jsonify()函数创建一个响应对象，并将映射信息转储到响应体中。

许多 Web 框架会显式地将一个 request 对象传递到代码中，但 Flask 与之不同，它

隐式地提供了一个全局 request 变量，这个变量指向当前的 request 对象。Flask 把传入的 HTTP 请求解析成 WSGI 环境字典，并利用这个字典创建这个对象。

这样的设计让视图代码更加简明：就像上例那样，如果服务器的响应不依赖于请求的内容，就没必要处理它。视图只需要确保返回了客户端应该获取的内容，并确保内容能被 Flask 序列化即可，一切都很简单。

在其他视图中，可直接导入并使用这个 request 变量。



在每个传入的请求中，变量 request 是全局唯一的，而且是线程安全的。Flask 使用一种称为“本地上下文”的机制，后面将讨论该机制。

接下来添加一些 print 方法，通过打印变量来看看底层发生了什么：

```
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/api')
def my_microservice():
    print(request)
    print(request.environ)
    response = jsonify({'Hello': 'World!'})
    print(response)
    print(response.data)
    return response
if
__name__ == '__main__':
    print(app.url_map)
    app.run()
```

运行修改后的代码，然后在其他 shell 中用 curl 命令发送请求，就能得到很多详细信息，如下所示：

```
$ python flask_details.py
Map([<Rule '/api' (GET, OPTIONS, HEAD) -> my_microservice>,
    <Rule '/static/<filename>' (GET, OPTIONS, HEAD) -> static>])
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)

<Request 'http://127.0.0.1:5000/api' [GET]>

{'wsgi.url_scheme': 'http', 'HTTP_ACCEPT': '*/*',
```

```
'wsgi.run_once': False, 'PATH_INFO': '/api', 'SCRIPT_NAME': '',
'wsgi.version': (1, 0), 'SERVER_SOFTWARE': 'Werkzeug/0.11.11',
'REMOTE_ADDR': '127.0.0.1',
'wsgi.input': <_io.BufferedReader name=5>,
'SERVER_NAME': '127.0.0.1', 'CONTENT_LENGTH': '',
'werkzeug.request': <Request 'http://127.0.0.1:5000/api' [GET]>,
'SERVER_PORT': '5000', 'HTTP_USER_AGENT': 'curl/7.51.0',
'wsgi.multiprocess': False, 'REQUEST_METHOD': 'GET',
'SERVER_PROTOCOL': 'HTTP/1.1', 'REMOTE_PORT': 22135,
'wsgi.multithread': False, 'werkzeug.server.shutdown': <function
   WSGIRequestHandler.make_environ.<locals>.shutdown_server at
   0x1034e12f0>,
'HTTP_HOST': '127.0.0.1:5000', 'QUERY_STRING': '',
'wsgi.errors': <_io.TextIOWrapper name='<stderr>' mode='w'
   encoding='UTF-8'>, 'CONTENT_TYPE': ''}

<Response 24 bytes [200 OK]>
b'{n "Hello": "World!"n}n'
127.0.0.1 - - [22/Dec/2016 15:07:01] "GET /api HTTP/1.1" 200
```

下面探索在这次调用中发生了什么：

- 路由匹配：Flask 创建了 Map 类。
- 请求：Flask 将一个请求对象传递给视图。
- 响应：一个返回给客户端的响应对象，包含了响应内容。

2.2.1 路由匹配

路由匹配发生在 app.url_map 中，它是 Werkzeug 中 Map 类的一个实例。该类使用正则表达式来判定被@app.route 装饰的函数是否与传入的请求匹配。路由匹配只会检查 route 调用里的路径参数，来判断函数是否匹配客户端的请求。

默认情况下，声明式路由仅接受 GET、OPTIONS 和 HEAD 方法的调用。如果在调用一个合法的调用点时，使用了不支持的 HTTP 方法，服务器会返回 405 Method Not Allowed 响应，并在 Allow 响应头中返回其所支持的 HTTP 方法列表。

```
$ curl -v -XDELETE localhost:5000/api
* Connected to localhost (127.0.0.1) port 5000 (#0)
> DELETE /api/person/1 HTTP/1.1
> Host: localhost:5000
> User-Agent: curl/7.51.0
```

```
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 405 METHOD NOT ALLOWED
< Content-Type: text/html
< Allow: GET, OPTIONS, HEAD
< Content-Length: 178
< Server: Werkzeug/0.11.11 Python/3.5.2
< Date: Thu, 22 Dec 2016 21:35:01 GMT
<
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>405 Method Not Allowed</title>
<h1>Method Not Allowed</h1>
<p>The method is not allowed for the requested URL.</p> *
   Curl_http_done: called premature == 0
   Closing connection 0
```

如果要支持特定的 HTTP 方法，那么可给路由装饰器添加 methods 参数，如下所示：

```
@app.route('/api', methods=['POST', 'DELETE', 'GET'])
def my_microservice():
    return jsonify({'Hello': 'World!'})
```



这里需要注意，由于请求处理器自动做了处理，OPTIONS 和 HEAD 方法被隐式添加到所有路由规则中。可通过在函数上设置 provide_automatic_options 为 False 从而停用这种行为。当需要在 OPTIONS 响应中加入自定义消息头时，这个做法简洁有效；例如，在处理跨域共享资源(CORS)问题时，需要添加诸如 Access-Control-Allow-*的消息头。

1. 变量和转换器

路由系统提供的另一个特性是能使用变量。

可通过<VARIABLE_NAME>语法来使用变量。这是一个标准的路由标识方法(Bottle 也在使用)，它允许使用动态值来描述 API 调用点。

例如，当创建一个函数来处理发送给/person/N的所有请求时(N是person的唯一ID)，可使用/person/<person_id>形式。

当 Flask 调用函数时，会转换 URL 中的值，将其作为参数值赋给 person_id：

```
@app.route('/api/person/<person_id>')
```

```
def person(person_id):
    response = jsonify({'Hello': person_id})
    return response

$ curl localhost:5000/api/person/3
{
    "Hello": "3"
}
```

如果多个路由匹配同一个 URL，路由映射器将使用特定规则集来确定应该调用哪个路由。在 Werkzeug 的路由模块中，是这样描述这个规则集的:

(1) 为提高性能，优先匹配没有任何参数的路由规则。因为我们期望能匹配得更快，而一些常见规则也不需要参数(例如索引页) 。

(2) 优先考虑更复杂的规则，所以第二个参数是权重的负数。

(3) 最终，用实际权重来排序。

这就是 Werkzeug 的规则，因此，权重用来对路由规则排序，但 Flask 并没有使用这种方式，也没有基于这种方式工作。简而言之，先选择参数较多的视图，然后按 Python 导入这些模块的顺序选择参数少的视图。一条经验法则是，务必使每个声明的路由在应用中都是唯一的，否则在判断使用哪个路由时会很棘手。

路由还包含基础转换器，可将变量转换成特定类型。例如，如果需要一个整型变量，那么可使用<int:VARIABLE_NAME>。对于 person 示例，可使用/person/<int:person_id>。

如果请求匹配一个路由，但转换器无法完成转换，除非还有其他路由能匹配相同路径，否则 Flask 会返回 404 错误。

内置转换器包括 string(默认转换器，转换成 Unicode 字符串)、int、float、path、any 和 uuid。

path 转换器类似于默认转换器，但包括斜杠/。它非常类似于正则表达式[^/].*?。

any 转换器允许组合多个值。它过于灵活，通常较少使用。uuid 转换器用于匹配 UUID 字符串。

创建自定义的转换器非常容易。例如，当需要将用户的 ID 与用户名匹配时，可创建一个转换器来查询数据库，将请求中的数字转换成用户名。

为此，只需要创建一个继承自 BaseConverter 的类即可。它需要实现两个方法：to_python()和 to_url()方法。前者将值转换成视图中用到的 Python 对象，后者则执行反

向操作(url_for()用到了 to_url()，下一节会讲到)。

```
from flask import Flask, jsonify, request
from werkzeug.routing import BaseConverter, ValidationError

_USERS = {'1': 'Tarek', '2': 'Freya'}
_IDS = {val: id for id, val in _USERS.items()}

class RegisteredUser(BaseConverter):
    def to_python(self, value):
        if value in _USERS:
            return _USERS[value]
        raise ValidationError()

    def to_url(self, value):
        return _IDS[value]

app = Flask(__name__)
app.url_map.converters['registered'] = RegisteredUser

@app.route('/api/person/<registered:name>')
def person(name):
    response = jsonify({'Hello hey': name})
    return response

if __name__ == '__main__':
    app.run()
```

转换失败时，会抛出 ValidationError 错误。路由映射器会认为请求与这个路由不匹配。

下面多调用几次，来分析实际工作方式：

```
$ curl localhost:5000/api/person/1
{
    "Hello hey": "Tarek"
}

$ curl localhost:5000/api/person/2
{
    "Hello hey": "Freya"
```

```
}

$ curl localhost:5000/api/person/3
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p> The requested URL was not found on the server. If you entered
   the URL manually please check your spelling and try again.</p>
```

不过要注意，本例的目的是演示转换器的威力。在真实应用中，注意不要过多依赖转换器。因为在代码的演进中，修改所有路由是非常痛苦的。



路由的最佳实践是：尽量让它保持静态和简单，将它当作函数上的标签即可。

2. url_for 函数

Flask 的路由系统中的最后一个有趣特性是 url_for()函数。给定一个视图，它将返回一个真实的 URL。

以下是前面应用中的一个例子：

```
>>> from flask_converter import app
>>> from flask import url_for
>>> with app.test_request_context():
...     print(url_for('person', name='Tarek'))
...
/api/person/1
```



上例使用“读取-求值-打印”循环(Read-Eval-Print Loop，REPL)的交互式环境。可直接运行 Python 可执行程序来启动。

若要在模板中展示一些视图的 URL，而且这些 URL 依赖于执行上下文，这个特性将非常有用。只需要将函数名指向 url_for 即可，而非硬编码这些链接。

2.2.2 请求

请求到来时，Flask 会在一个线程安全的代码块里调用视图，并使用 Werkzeug 的局部助手程序 local(http://werkzeug.pocoo.org/docs/latest/local/)。这个助手程序的工作方式与 Python 中的 threading.local(https://docs.python.org/3/library/threading.html#thread-local-data)类似，它确保每个线程里都有一个特定于请求的独立环境。

换句话说，在视图中访问全局的请求对象时，可保证这个对象在当前线程里是唯一的。在多线程环境下，它不会将数据泄露给其他线程。

如前所述，Flask 使用传入的 WSGI 环境数据来创建请求对象。这个对象是 request 类的实例，合并了若干个 mixin 类，负责解析传入环境变量中的特定请求头。



要了解关于 WSGI 环境的更多细节，可访问 WSGIPEP(Python Environment Proposal)，见 https://www.python.org/dev/peps/pep-0333/#environ-variables。

最重要的是，不需要解析工作，视图即可通过 request 对象的属性来检视(introspect)传入的请求。Flask 特别擅长完成这部分工作。例如，当 Flask 发现传入了 Authorization 请求头时，会自动解析它。

在下例中，客户端会向服务器发送 HTTP Basic Auth，并编码为 base64 形式。Flask 检测到 Basic 的前缀，进行解析，并存入 request.authorization 属性中的 username 和 password 字段内。

```
from flask import Flask, request

app = Flask(__name__)

@app.route("/")
def auth():
    print("The raw Authorization header")
    print(request.environ["HTTP_AUTHORIZATION"])
    print("Flask's Authorization header")
    print(request.authorization)
    return ""

if __name__ == "__main__":
    app.run()

$ curl http://localhost:5000/ -u tarek:password

$ bin/python flask_auth.py
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
The raw Authorization header
Basic dGFyZWs6cGFzc3dvcmQ=
Flask's Authorization header
{'username': 'tarek', 'password': 'password'}
127.0.0.1 - - [26/Dec/2016 11:33:04] "GET / HTTP/1.1" 200 -
```

这样的行为可轻松地在 request 对象上实现插件式身份验证系统。

其他常见的请求元素(如 cookie、文件等)都可通过其他属性来访问，见后续章节的介绍。

2.2.3 响应

上例用到了 jsonify()函数，它使用视图返回的映射来创建一个响应对象。

从技术角度看，响应对象是一个可直接使用的标准 WSGI 应用程序。Flask 对其进行封装，它与 WSGI 的 environ 一起被调用，Web 服务器会接收 start_response 函数。

当 Flask 通过 URL 映射选择一个视图时，它期望视图能返回一个可接收 environ 和 start_response 参数的可调用对象。



这个设计看似有些奇怪，因为当使用 WSGI environ 再次调用 response 对象时，WSGI environ 已被解析为 request 对象。但实际上，这只是实现细节。当代码需要与请求交互时，可使用全局的 request 对象，并忽略 response 类的细节。

当返回值不是可调用对象时，Flask 会尝试将下列类型的返回值转换成 response 对象：

- **str**：数据会被编码成 UTF-8 的字符串，并用作 HTTP 响应体。
- **bytes/bytesarray**：用作响应体。
- **(response,status,headers)元组**：response 可以是 response 对象或上述类型之一。status 是一个 integer 类型的值，它会重写响应的状态码，headers 是扩展响应头的映射。
- **(response, status)元组**：与前一种情况类似，但没有特定的响应头。
- **(response, headers)元组**：与前面的情况类似，只包含额外的响应头。

其他情况下会抛出一个异常。

大多数情况下，当构建微服务时，通常使用内置的 jsonify()函数。但如果要在调用点返回其他类型的内容，那么可很容易地创建一个函数，把生成的数据转换到 response 类中。

以下是一个返回 YAML 格式内容的例子。yamlify()函数返回一个(response, status, headers)元组，Flask 将其转换成合适的 response 对象：

```
from flask import Flask
import yaml # requires PyYAML

app = Flask(__name__)
```

```
def yamlify(data, status=200, headers=None):
    _headers = {'Content-Type': 'application/x-yaml'}
    if headers is not None:
        _headers.update(headers)
    return yaml.safe_dump(data), status, _headers

@app.route('/api')
def my_microservice():
    return yamlify(['Hello', 'YAML', 'World!'])
if __name__ == '__main__':
    app.run()
```

Flask 处理请求的方式，可总结为以下几步：

(1) 启动应用时，将所有被@app.route()装饰的函数注册为视图，并保存在 app.url_map 中。

(2) 根据其调用点和方法，请求被分发到合适的视图上。

(3) 在一个线程安全和线程局部的执行上下文中，创建 request 对象。

(4) 返回一个封装响应内容的 response 对象。

这 4 个步骤概括了使用 Flask 构建应用所需理解的一切。下一节将概述 Flask 中最重要的内置特性，这些特性与请求-响应机制相关。

2.3　Flask 的内置特性

上一节详细解释了 Flask 如何处理请求，这些知识对于快速上手 Flask 已经足够了。Flask 还自带了很多非常有用的助手程序。这一节会介绍几个主要程序：

- **Session 对象**：基于 cookie 的数据。
- **全局值(Globals)**：在请求上下文中存储数据。
- **信号(Signals)**：发送和截取事件。
- **扩展和中间件**：添加功能。
- **模板(Template)**：构建基于文本的内容。
- **配置(Configuring)**：在配置文件中将启动的选项进行分组。
- **Blueprint**：使用名称空间组织代码。
- **错误处理和调试**：处理应用中的错误。

2.3.1 Session 对象

与 request 对象类似，Flask 创建了 Session 对象，它在请求上下文中是唯一的。

Session 对象是一个类似于字典的对象，Flask 将其序列化成 cookie，并发给用户。包含在会话映射中的数据会被转储到 JSON 格式的映射中，然后 Flask 使用 zlib 对其压缩，最终使用 base64 进行编码。

会话被序列化后，itsdangerous 库(https://pythonhosted.org/itsdangerous/)对其内容进行签名，签名时使用在应用级别定义的 secret_key。签名会使用 HMAC(https://en.wikipedia.org/wiki/Hash-based_message_authentication_code)和 SHA1 算法。

签名(会在数据中添加后缀)可确保不知道密钥的客户端不能篡改存储在 cookie 中的数据。注意这里的数据本身并未加密。

Flask 允许自定义签名算法，但对于在 cookie 存储数据的场景，使用 HMAC+SHA1 已经足够了。

然而，如果构建的微服务不返回 HTML，将很少依赖 cookie，因为 cookie 专用于 Web 浏览器。但通过为每个用户提供一个易变的键值存储，有助于加快完成服务器端的一部分工作。例如，如果在用户每次访问时，都需要执行一些数据库查询来获取用户信息，那么使用类似 Session 的对象将信息缓存在服务器端可提高处理速度。

2.3.2 全局值

如前所述，Flask 提供一种存储全局变量的机制，这种机制保证这些全局变量在特定线程和请求上下文中是唯一的。这个机制用在 request 和 session 上，也可存储其他自定义对象。

flask.g 对象包含所有全局值，可在它上面给任何属性赋值。

在 Flask 中使用@app.before_request 装饰器装饰一个方法时，在每个请求被分发到视图前，应用会调用这个方法。

利用 before_request 来设置全局值是 Flask 的一个典型模式。通过这个模式，在请求上下文中调用的所有方法都能与 g 交互并获取数据。

如下所示，当客户端进行 HTTP 基本身份验证时，会将提供的 username 复制到 g.user 属性中。

```
from flask import Flask, jsonify, g, request

app = Flask(__name__)
```

```
@app.before_request
def authenticate():
    if request.authorization:
        g.user = request.authorization['username']
    else:
        g.user = 'Anonymous'

@app.route('/api')
def my_microservice():
    return jsonify({'Hello': g.user})

if __name__ == '__main__':
    app.run()
```

这样，当客户端请求/api 视图时，authenticate()函数会根据所提供的请求头来设置 g.user。

```
$ curl http://127.0.0.1:5000/api
{
    "Hello": "Anonymous"
}
$ curl http://127.0.0.1:5000/api --user tarek:pass
{
    "Hello": "tarek"
}
```

当需要在代码中共享数据，而且数据特定于请求上下文时，都可通过 flask.g 来实现。

2.3.3　信号

Flask 集成了 Blinker 库(https://pythonhosted.org/blinker/)。它是一个处理信号的库，可为事件订阅一个函数。

事件是 blinker.signal 类的实例，被创建时有一个唯一标签。Flask 在 0.12 版本中有数十个这样的实例。Flask 在处理请求时，会在特定时刻触发信号。可参考 http://flask.pocoo.org/docs/latest/api/#core-signals-list 查看完整列表。

要注册特定事件，可调用信号的 connect 方法。当代码调用了信号的 send 方法时，会触发信号。send 方法接收到的额外参数用来给所有已注册的函数传递数据。

在下例中，将 finished 函数注册到 request_finished 信号上，这个函数将接收到 response 对象：

```
from flask import Flask, jsonify, g, request_finished
from flask.signals import signals_available

if not signals_available:
    raise RuntimeError("pip install blinker")
app = Flask(__name__)

def finished(sender, response, **extra):
    print('About to send a Response')
    print(response)

request_finished.connect(finished)

@app.route('/api')
def my_microservice():
    return jsonify({'Hello': 'World'})

if __name__ == '__main__':
    app.run()
```

注意，安装 Blinker 后才可使用信号机制。安装 Flask 时，并未将其作为默认的依赖项来安装。

Flask 中一些信号的实现在微服务中没多大用处，例如当框架渲染模板时所发生的信号。但在 Flask 触发的信号中，一些有趣的信号贯穿整个请求周期，可用来记录日志。

例如，当异常发生但框架尚未处理时，会触发 got_request_exception 信号。Sentry(https://sentry.io)的 Python 客户端(Raven)就利用这个信号，将自己变成探针来记录异常。

如果期望在应用中使用事件来触发一些自定义特性，同时降低代码的耦合性，那么实现自定义信号会很有意义。

例如，假定有一个生成 PDF 报表的微服务，这个报表会使用密码算法来生成签名，那么可触发 report_ready 信号，并事先将签名的方法注册到这个事件上。

Blinker 实现中的一个重要方面是，在调用所有已注册的函数时，不会遵循特定顺序来执行，并且它们在 signal.send 方法上是异步执行的。所以，如果应用开始使用大

量信号，那么在处理请求时，所有信号的触发过程会耗费大量时间，进而导致性能瓶颈。

如果期望信号工作时不影响响应的返回，可考虑使用诸如 RabbitMQ(https://www.rabbitmq.com/)的队列工具，让任务排队等候，由一个独立服务来处理队列。

2.3.4 扩展和中间件

Flask 扩展其实只是简单的 Python 项目，安装后会提供一个以 flask_something 方式命名的包或模块。在早期版本中，命名方式是 flask.ext.something。

扩展项目必须遵循一些准则，详见 http://flask.pocoo.org/docs/latest/extensiondev 中的描述。或多或少地，这些准则都可作为任何 Python 项目的良好实践。Flask 有一个精心整理的扩展列表，详见 http://flask.pocoo.org/extensions/。当寻找一些额外特性时，应该首先查看这个列表。扩展提供的功能取决于它的开发者，除了 Flask 文档中提到的准则外，没有太多强制性要求。

另一种扩展 Flask 的机制是使用 WSGI 中间件。WSGI 中间件是一种扩展 WSGI 应用的模式，这种模式通过封装对 WSGI 调用点的请求来实现。

下例中的中间件伪造了 X-Forwarded-For 消息头，让 Flask 应用以为自己位于反向代理(诸如 nginx)之后。在测试环境中，当应用尝试获取远程 IP 地址时，我们希望确保应用能正确运行，这个中间件用处很大。因为 remote_addr 属性返回的是代理的 IP，而非客户端的真实 IP。

```
from flask import Flask, jsonify, request
import json

class XFFMiddleware(object):
    def __init__(self, app, real_ip='10.1.1.1'):
        self.app = app
        self.real_ip = real_ip
    def __call__(self, environ, start_response):
        if 'HTTP_X_FORWARDED_FOR' not in environ:
            values = '%s, 10.3.4.5, 127.0.0.1' % self.real_ip
            environ['HTTP_X_FORWARDED_FOR'] = values
        return self.app(environ, start_response)

app = Flask(__name__)
app.wsgi_app = XFFMiddleware(app.wsgi_app)
```

```
@app.route('/api')
def my_microservice():
    if "X-Forwarded-For" in request.headers:
        ips = [ip.strip() for ip in
            request.headers['X-Forwarded-For'].split(',')]
        ip = ips[0]
    else:
        ip = request.remote_addr
    return jsonify({'Hello': ip})

        if __name__ == '__main__':
    app.run()
```



注意，此处使用 app.wsgi_app 来封装 WSGI 应用。如前所述，在 Flask 中，应用对象并不是 WSGI 应用。

在应用访问 WSGI environ 前修改它的内容是没有问题的。但如果在 WSGI 中间件修改响应，会导致开发工作变得非常痛苦。

WSGI 协议规定，在应用返回真正的响应体内容前，会以响应的状态码和消息头作为参数调用 start_response 函数。遗憾的是，应用上的单个函数调用会触发这个两步机制。因此，为在应用外修改响应结果，必须使用一些回调魔法。

一个很好的例子是，当修改响应的消息体时，会影响消息头 Content-Length 的值。因此需要中间件拦截应用发送的消息头，并在消息体被修改后，重写消息头的值。

上面只是 WSGI 协议设计上的问题之一，围绕它还有很多其他问题。

除非你希望开发的功能在其他 WSGI 框架上也能运行，否则没必要使用 WSGI 中间件来扩展应用。编写一个 Flask 扩展是更值得推荐的方案，因为扩展可在 Flask 应用内部进行交互。

2.3.5 模板

返回 JSON 或 YAML 格式的文档非常简单，这是因为我们仅序列化了数据。大部分微服务会生成能被机器转换的数据。但某些情况下，需要创建一些包含布局的文档——例如 HTML 页面、PDF 报表或 Email。

为处理文本，Flask 集成了 Jinja 模板引擎(http://jinja.pocoo.org)。Flask 并入 Jinja 主要是用来生成 HTML 文档。通过诸如 render_template 的助手程序，可利用 Jinja 模板和传入的数据来生成响应体。

不过，Jinja 不仅用来生成 HTML 或其他基于标签的文档，它可创建任何基于文本

的文档。

例如，如果一些微服务会发送邮件，但使用标准库中的 email 包来生成邮件内容非常繁杂，此时可选择 Jinja。

下面是一个创建邮件模板的例子：

```
Date: {{date}}
From: {{from}}
Subject: {{subject}}
To: {{to}}
Content-Type: text/plain

Hello {{name}},

We have received your payment!

Below is the list of items we will deliver for lunch:

{% for item in items %}- {{item['name']}} ({{item['price']}} Euros)
{% endfor %}

Thank you for your business!

--
Tarek's Burger
```

Jinja用双大括号来标记那些会被替换成值的变量。变量可以是程序执行中传给Jinja的任何东西。

也可在模板中直接使用 Python 中的 if 和 for 代码块，使用{% for x in y % }...{% endfor %}和{% if x %}...{% endif %}来标记。

以下的 Python 脚本使用 email 模板生成一个完整且有效的 RFC 822 消息，可通过 SMTP 来发送它。

```
from datetime import datetime
from jinja2 import Template
from email.utils import format_datetime

def render_email(**data):
    with open('email_template.eml') as f:
        template = Template(f.read())
```

```
    return template.render(**data)

data = {'date': format_datetime(datetime.now()),
        'to': 'bob@example.com',
        'from': 'tarek@ziade.org',
        'subject': "Your Tarek's Burger order",
        'name': 'Bob',
        'items': [{'name': 'Cheeseburger', 'price': 4.5},
                  {'name': 'Fries', 'price': 2.},
                  {'name': 'Root Beer', 'price': 3.}]}

print(render_email(**data))
```

render_email 函数使用 Template 类，它根据给定方法生成邮件内容。

Jinja 非常强大，包含很多功能。但由于不在本章的讨论范围内，因此不会展开介绍。如果需要在微服务中完成一些与模板相关的工作，Jinja 将是一个好选择，而且 Flask 已经包含了它。可在 http://jinja.pocoo.org/docs 查看 Jinja 功能的完整文档。

2.3.6 配置

构建应用时，需要公开一些运行选项，例如数据库的连接信息，或在部署中才使用的一些变量。

Flask 使用与 Django 类似的配置管理机制。Flask 对象包含一个 Config 对象，这个对象包含一些内置变量，在启动 Flask 应用时，可通过自定义的配置对象来更新。

例如，可在 prod_settings.py 文件中定义一个 Config 类，如下所示：

```
class Config:
    DEBUG = False
    SQLURI = 'postgres://tarek:xxx@localhost/db'
```

然后，在 app 对象中，通过 app.config.from_object 加载它：

```
>>> from flask import Flask
>>> app = Flask(__name__)
>>> app.config.from_object('prod_settings.Config')
>>> print(app.config)
<Config {'SESSION_COOKIE_HTTPONLY': True, 'LOGGER_NAME': '__main__',
```

```
'APPLICATION_ROOT': None, 'MAX_CONTENT_LENGTH': None,
'PRESERVE_CONTEXT_ON_EXCEPTION': None,
'LOGGER_HANDLER_POLICY': 'always',
'SESSION_COOKIE_DOMAIN': None, 'SECRET_KEY': None,
'EXPLAIN_TEMPLATE_LOADING': False,
'TRAP_BAD_REQUEST_ERRORS': False,
'SESSION_REFRESH_EACH_REQUEST': True,
'TEMPLATES_AUTO_RELOAD': None,
'JSONIFY_PRETTYPRINT_REGULAR': True,
'SESSION_COOKIE_PATH': None,
'SQLURI': 'postgres://tarek:xxx@localhost/db',
'JSON_SORT_KEYS': True, 'PROPAGATE_EXCEPTIONS': None,
'JSON_AS_ASCII': True, 'PREFERRED_URL_SCHEME': 'http',
'TESTING': False, 'TRAP_HTTP_EXCEPTIONS': False,
'SERVER_NAME': None, 'USE_X_SENDFILE': False,
'SESSION_COOKIE_NAME': 'session', 'DEBUG': False,
'JSONIFY_MIMETYPE': 'application/json',
'PERMANENT_SESSION_LIFETIME': datetime.timedelta(31),
'SESSION_COOKIE_SECURE': False,
'SEND_FILE_MAX_AGE_DEFAULT': datetime.timedelta(0, 43200)}>
```

不过，使用 Python 模块作为配置文档存在两个缺点：

首先，除了一些简单和扁平的类外，开发过程中可能在配置模块中添加更复杂的代码。这样，随着开发的进行，这些配置代码将与其他应用代码没有太大区别。但在部署时，通常不会发生分开管理配置文件与代码的情况。

其次，如果有其他团队负责管理应用的配置文件，也需要编辑 Python 代码。这导致更容易引入问题。例如，在 Python 模块外创建 Puppet 模板，会比创建扁平静态的配置文档更困难。

由于 Python 通过 app.config 来公开配置信息，因此从 YAML 或其他基于文本的文件中加载其他选项也变得简单。

INI 格式是 Python 社区中最常用的格式。这是因为标准库包含一个 INI 解析器，而且非常通用。

有很多 Flask 扩展可从 INI 文件中加载配置信息，但使用标准库中的 ConfigParser 会很麻烦。尽管如此，使用 INI 文件需要特别注意：INI 文件中的变量都是字符串，应用需要小心地将其转换成正确类型。

konfig项目(https://github.com/mozilla-services/konfig)是ConfigParser上的小封装层，能自动转换一些简单类型，如整型和布尔型。

在 Flask 中使用 konfig 很简单：

```
$ more settings.ini
[flask]
DEBUG = 0
SQLURI = postgres://tarek:xxx@localhost/db

$ python
>>> from konfig import Config
>>> from flask import Flask
>>> c = Config('settings.ini')
>>> app = Flask(__name__)
>>> app.config.update(c.get_map('flask'))
>>> app.config['SQLURI']
'postgres://tarek:xxx@localhost/db
```

2.3.7 Blueprint

当微服务包含多个调用点时，最终会有大量被装饰的方法——或许每个调用点都有好几个方法。整理和组织代码的第一个逻辑步骤是，为每个调用点使用一个模块。在创建应用实例中，要确保导入它们，以便 Flask 能注册其中的视图。

例如，如果一个微服务管理一个公司的雇员数据库，可使用一个调用点与所有雇员交互，另一个调用点和团队交互。可使用以下 3 个模块来管理应用：

- app.py：包含 Flask 的 app 对象，用它来启动应用。
- employees.py：提供与雇员相关的所有视图。
- teams.py：提供与团队相关的所有视图。

这样，雇员和团队可当作应用的子集，各自有一些特定的工具类和配置。

Blueprint 更强化了这个逻辑，它提供一种方式，将视图按不同名称空间分组。可创建与 Flask 应用对象类似的 Blueprint 对象，然后用它来组织视图。在应用的初始化过程中，使用 app.register_blueprint 来注册它们，这样就保证了 Blueprint 中定义的所有视图成为应用的一部分。

下面是雇员 Blueprint 的一种可能实现方式：

```
from flask import Blueprint, jsonify

teams = Blueprint('teams', __name__)

_DEVS = ['Tarek', 'Bob']
_OPS = ['Bill']
```

```
_TEAMS = {1: _DEVS, 2: _OPS}

@teams.route('/teams')
def get_all():
    return jsonify(_TEAMS)

@teams.route('/teams/<int:team_id>')
def get_team(team_id):
    return jsonify(_TEAMS[team_id])
```

主模块(app.py)导入这个文件后，可用 app.register_blueprint(teams)来注册这个 Blueprint。

当希望在其他应用中复用一个通用视图集合，或在同一应用中多次使用某个通用视图集合时，Blueprint 机制会很有趣。

例如，Flask-Restless(https://flask-restless.readthedocs.io)扩展提供了创建、读取、更新和删除资源的工具，它检视 SQLAlchemy 的模型，针对每个模型生成一个 Blueprint，这样就自动通过 REST API 公开了数据库。

下例来自 Flask-Restless 文档(Person 是 SQLAlchemy 模型)：

```
blueprint = manager.create_api_blueprint(Person, methods=['GET',
'POST'])
app.register_blueprint(blueprint)
```

2.3.8　错误处理和调试

当应用出现错误时，重要的是能控制客户端收到的响应。在返回 HTML 的 Web 应用中，当遇到 404 或 50x 错误时，通常会指定特定 HTML 页面，这也是 Flask 开箱即用的工作方式。但在构建微服务时，你需要更多地控制返回给客户端的内容——这正是自定义错误处理程序的用处。

Flask 的另一个重要特性是，它允许在发生意外错误时调试代码。本节将探讨 Flask 内置的调试器；当应用以调试模式(debug mode)运行时，就会激活调试器。

1. 自定义错误处理程序

当代码不处理异常时，Flask 返回的 500 错误不包含任何详细信息(诸如异常跟踪记录)。返回一个通用错误是安全的默认行为，这种做法可避免通过错误体向用户泄露任何私密信息。

默认的 500 错误响应是一个简单的 HTML 页面以及对应的状态码：

```
$ curl http://localhost:5000/api
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>500 Internal Server Error</title>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error and was unable to complete your
request. Either the server is overloaded or there is an error in the
application.</p>
```

使用 JSON 实现微服务时，一个良好实践是确保返回给客户端的响应(包括任何异常)都是 JSON 格式。微服务的消费者会期望每个响应都可被机器解析。

Flask 提供了一些函数，让你可自定义处理错误程序。首先是@app.errorhandler 装饰器，它的工作方式与@app.route 相似。不过这个装饰器将函数链接到错误码上，而不是提供一个 API 调用点。

下例使用@app.errorhandler 来连接一个函数，当 Flask 返回 500 服务器响应时(出现任何代码异常时)，这个函数返回 JSON 格式的错误。

```
from flask import Flask, jsonify

app = Flask(__name__)

@app.errorhandler(500)
def error_handling(error):
    return jsonify({'Error': str(error)}, 500)

@app.route('/api')
def my_microservice():
    raise TypeError("Some Exception")

if __name__ == '__main__':
    app.run()
```

不论代码抛出哪种异常，Flask 都会调用这个错误视图。

然而，如果应用发生了 HTTP 404 和其他 4*xx*、50*x* 错误，Flask 依然返回默认的 HTML 格式响应。

为确保应用能针对每个 4*xx* 和 50*x* 返回 JSON 格式响应，需要将函数注册到每个错误码上。

在 abort.mapping 字典中，可找到所有错误的列表。在下面的代码片段中，使用 app.register_error_handler 将 error_handling 方法注册到所有错误上。app.register_error_

handler 的效果与@app.errorhandler 装饰器类似：

```
from flask import Flask, jsonify, abort
from werkzeug.exceptions import HTTPException, default_exceptions

def JsonApp(app):
    def error_handling(error):
        if isinstance(error, HTTPException):
            result = { 'code': error.code, 'description':
                       error.description, 'message': str(error)}
        else:
            description = abort.mapping[500].description
            result = {'code': 500, 'description': description,
                     'message': str(error)}

        resp = jsonify(result)
        resp.status_code = result['code']
        return resp

    for code in default_exceptions.keys():
        app.register_error_handler(code, error_handling)

    return app

app = JsonApp(Flask(__name__))

@app.route('/api')
def my_microservice():
    raise TypeError("Some Exception")

if __name__ == '__main__':
    app.run()
```

JsonApp 函数封装一个 Flask 应用实例，针对所有可能发生的 4*xx* 和 50*x* 错误，设置了自定义错误处理程序。

2. 调试模式

Flask 应用的 run 方法有一个 debug 选项。启用后会在调试模式下运行应用。

```
app.run(debug=True)
```

调试模式是一个特殊模式。内置调试器会拦截任何错误，并允许你在浏览器中与应用交互。

web-debugger 中的控制台允许你与当前应用交互并检视变量，或在当前的执行帧中执行任何 Python 代码。

Flask甚至允许配置第三方调试器。例如，JetBrains的PyCharm(https://www.jetbrains.com/pycharm)是一个针对 Python的商业 IDE，提供了强大的可视化调试器，设置后可与Flask一起运行。



在调试模式时，只有提供 PIN 才能访问控制台(如图 2-1 所示)；但调试模式允许远程执行代码，因此仍然存在安全隐患。2015 年，黑客就通过 Flask 调试器入侵了 Patreon 在线服务。需要特别注意，不要在生产环境下使用调试模式！安全分析工具 Bandit(https://wiki.openstack.org/wiki/Security/Projects/Bandit)能跟踪那些使用了普通调试标记的 Flask 应用，从而防止使用这个标记部署应用。

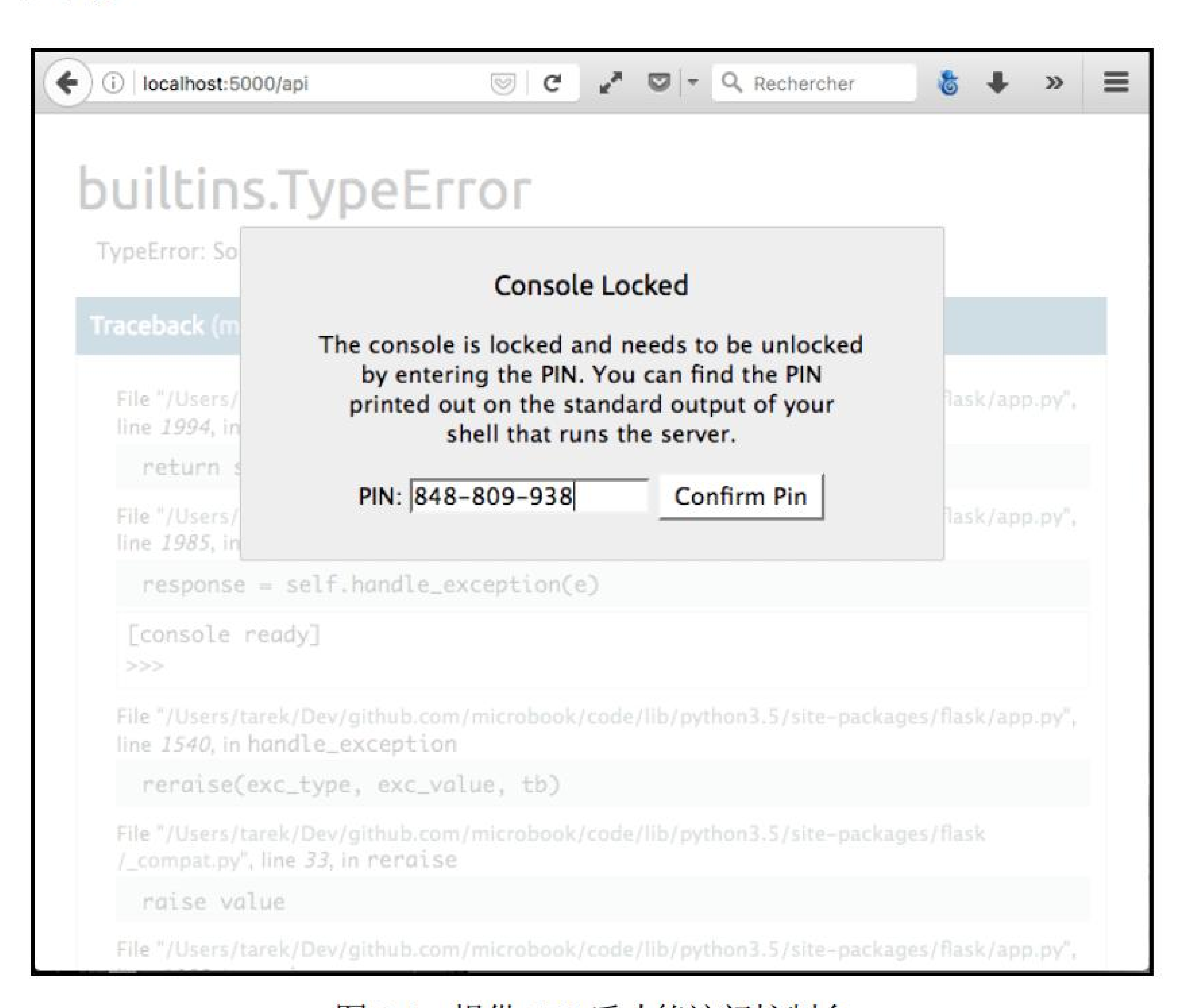


图 2-1　提供 PIN 后才能访问控制台

当跟踪问题时，使用普通 pdb 模块在代码中插入 pdb.set_trace()也是一种不错的方法。

2.4　微服务应用的骨架

前面介绍了 Flask 的工作方式和大部分内置特性(本书后面将使用这些特性)。

还有一个未涉及的话题是如何在项目中组织代码，以及如何实例化 Flask 应用。到目前为止，所有例子都使用单独 Flask 模块，调用 app.run()来运行服务。

当然，除非代码只有几行，否则在一个模块中编写所有代码是很糟糕的。由于要发布和部署代码，最好将代码放在一个 Python 包中，这样就能使用标准的包管理工具(如 pip 和 Setuptools)安装它。

一个好方法是用 Blueprint 组织视图代码，每个 Blueprint 包含一个模块。

最后，由于 Flask 提供一个通用运行器(runner)来查找 app 变量，而这个 app 变量是通过在模块上指定 FLASK_APP 环境变量来确定的，因此可从代码中删除对 run()方法的调用。使用这个运行器时，可指定一些额外选项，例如配置应用运行时使用的主机名和端口等。

GitHub上有一个为本书创建的微服务项目(https://github.com/Runnerly/microservice)。它是一个通用Flask项目，可用来启动一个微服务应用。它实现了简单的代码布局，适于构建微服务。

你可安装并运行它，然后修改它。



这个项目使用 flakon(https://github.com/Runnerly/flakon)。flakon 是一个极简的助手程序，负责使用 INI 文件和默认 JSON 行为，来配置和实例化 Flask 应用。flakon 也是为本书创建的，目的是使用最少样板代码，让你专注于构建微服务。flakon 并非是强制的，如果一些技术决策不适合你，你可将它从项目中删除，然后开发函数来创建应用，或使用提供了这种功能的开源项目。

上述 microservice 项目的骨架包含以下结构：

- setup.py：Distutil 的安装文件，用来安装和发布应用。
- Makefile：包含用来构建和运行项目的任务。
- settings.ini：INI 格式的应用默认配置文件。
- requirements.txt：遵循 pip 格式的项目依赖项。
- myservices/：真正的应用包。
    - __init__.py。
    - app.py：app 模块，包含应用本身。
    - views/：以 Blueprint 组织的视图。
        - __init__.py。

    - home.py：home blueprint，处理根调用点。
- tests：包含所有测试的目录。
    - __init__.py。
    - test_home.py：home blueprint 中视图的测试。

在以下代码中，app.py 文件通过 flakon 的 create_app 助手方法实例化一个 Flask 应用，并使用一些参数，如注册的 Blueprint 列表。

```
import os
from flakon import create_app
from myservice.views import blueprints

_HERE = os.path.dirname(__file__)
_SETTINGS = os.path.join(_HERE, '..', 'settings.ini')

app = create_app(blueprints=blueprints, settings=_SETTINGS)
```

home.py 视图使用 flakon 的 JsonBlueprint 类，实现了前一节提到的错误处理方式。当视图返回字典对象时，会自动调用 jsonify()函数——就像 Bottle 框架那样：

```
from flakon import JsonBlueprint

home = JsonBlueprint('home', __name__)

@home.route('/')
def index():
    """Home view.

    This view will return an empty JSON mapping.
    """
    return {}
```

可通过 Flask 内置的命令行工具，使用包名，来运行上面的示例应用：

```
$ FLASK_APP=myservice flask run
 * Serving Flask app "myservice"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

从现在开始，如果要给微服务应用创建 JSON 视图，需要在 microservice/views 中添加模块，并在 microservice/tests 添加对应的测试。

2.5　本章小结

本章详述 Flask 框架，以及如何用它构建微服务。

本章要点如下：

- Flask 通过 WSGI 协议包装一个简单的请求响应机制，允许使用普通的 Python 代码来编写应用。
- Flask 易于扩展，能运行在 Python 3 上。
- Flask包含一些好用的内置特性：Blueprint、全局值、信号、模板引擎、错误处理程序和调试器。
- microservice 项目是一个 Flask 骨架，本书用它来编写微服务。这个简单的应用使用 INI 作为配置，确保应用以 JSON 格式返回响应。

第 3 章将关注开发方法论：如何持续地编码、测试和写文档。

第 3 章

良性循环：编码、测试和写文档

每个已部署的软件项目都会受到难以避免的 bug 的折磨；为消除 bug，需要持续投入时间和金钱。

有一种在编码时同时编写测试的方法叫做 TDD(Test-Driven Development，测试驱动的开发)，使用 TDD 虽然并非总能提高项目质量，但能让团队更敏捷。当开发者需要修复一个 bug，或对应用进行重构时，基于测试将可更快捷、更好地完成工作。如果团队中有人破坏了一个功能，测试也将立即提醒团队注意。

编写测试非常耗时，但从长远看，这是一个能让项目健康成长的最佳方法。当然，总可能编写差劲的测试并最终导致糟糕的结果，或创建的测试套件极难维护而且运行时间过长。世界上最好的工具和流程并不会阻止马虎的开发者编写出差劲的软件，如图 3-1 所示。

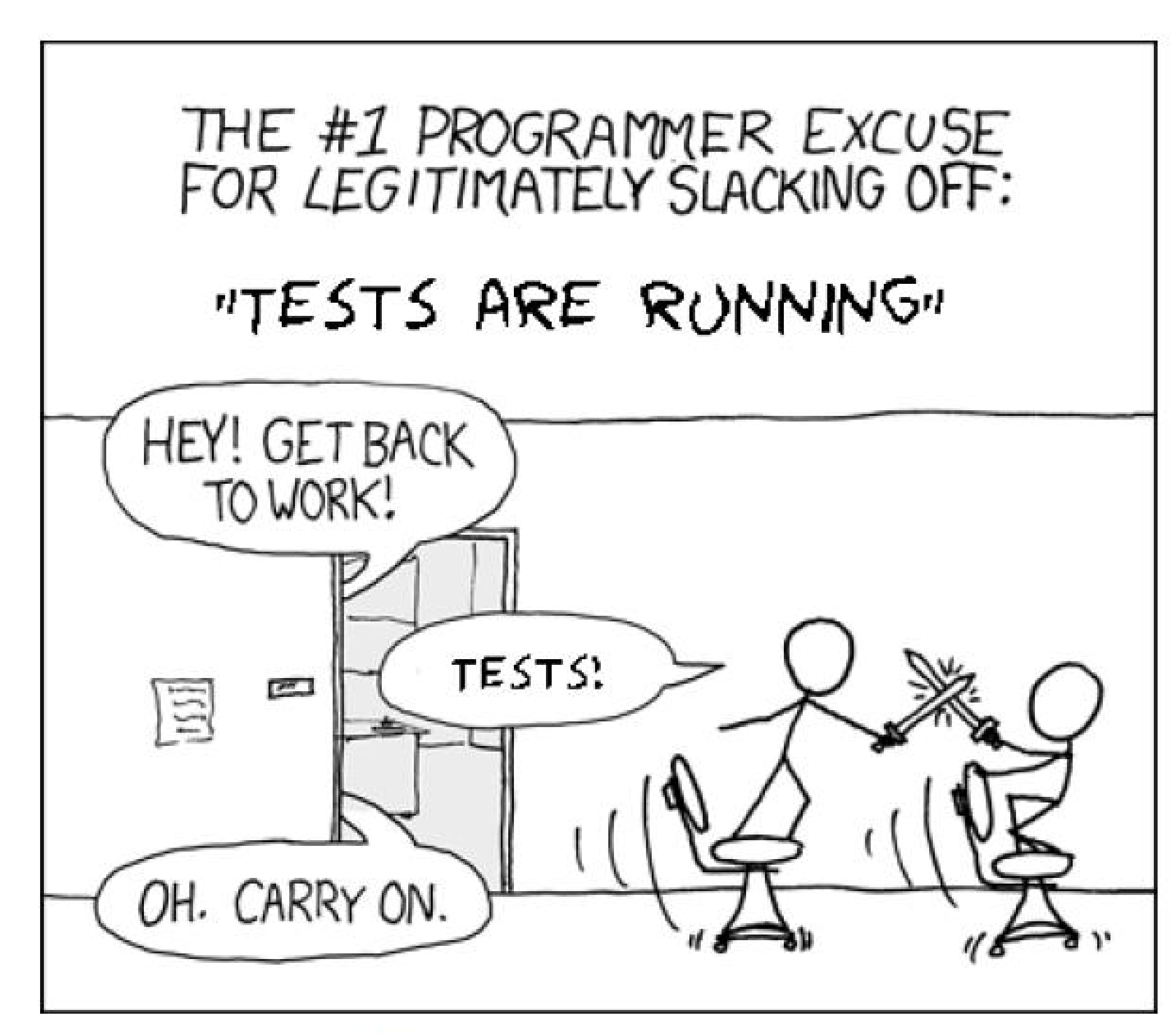


图 3-1　凡事贵在用心

长期以来，软件行业一直在讨论 TDD 的功效。但最近十年，通过量化 TDD 的好处，更多的研究论文得出结论：TDD 从长期看花钱更少，质量更高。你可在 http://biblio.gdinwiddie.com/biblio/StudiesOfTestDrivenDevelopment 页面上看到关于此主题的很多研究论文。

编写测试也是一种从不同角度查看代码的好方法。设计的 API 是否合理？代码是否恰当地结合在一起？当团队扩编和发生变化时，测试是最好的信息来源。不同于文档，测试反映了当前代码版本的行为。

即便维护文档很难也很耗时，但文档记录依然是项目中重要的一部分。当任何人开始使用你的软件，或加入团队开始工作时，这是他的第一站。应用是如何被安装和配置的？如何运行测试或添加新功能？软件是如何设计的，为什么这样设计？

一段时间后，除非安排专人，否则极少看到一个项目的文档会随着代码及时更新。令开发者备感沮丧的是，发现文档中的代码示例在重构后被破坏了。不过有一些方法可缓解这些问题；例如，在文档中提取的代码可作为测试套件的一部分，由测试套件确保这些代码是可工作的。

任何场景下，无论你在编写测试和文档上耗费了多少精力，有一个黄金法则：在项目中应该持续地写测试、写文档，和写代码。换言之，在理想情况下，代码中的更改最好能立即反应在测试和文档中。

提供了一些如何在 Python 中编写测试的一般性提示后，本章将重点讨论在使用 Flask 构建微服务时，有哪些编写测试和文档的工具可供使用，并介绍如何用主流的在线服务来实现持续集成。

这些内容分为 5 个部分：

- 各种测试类型的差异
- 使用 WebTest 测试微服务
- 使用 pytest 和 Tox
- 开发者文档
- 持续集成

3.1 各种测试类型的差异

测试有很多种类型，有时当我们想知道讨论的是哪种测试时，会感到困惑。例如，当提到功能测试时，不同的项目性质可能代表不同类型的测试。

在微服务领域，可将测试按照 5 个不同目标进行分类：

- 单元测试(unit test)：在隔离场景下，确保类或函数运行的结果和期望一致

- 功能测试(functional test)：从微服务消费者视角，验证微服务的行为和期望一致，并且即使请求不当，依然能正常工作。
- 集成测试(integration test)：验证微服务如何与它的所有网络依赖项进行集成。
- 负载测试(load test)：度量微服务的性能。
- 端到端测试(end-to-end test)：用端到端测试验证整个系统。

下面将详细介绍这些内容。

3.1.1 单元测试

单元测试是添加到项目中的最简单测试，Python 标准库附带了编写单元测试所需的内容。在一个基于 Flask 的项目中，通常都围绕视图(一些函数和类)进行独立的单元测试。

然而在 Python 项目中，“分离(separation)”概念是非常含糊的。这是因为没有使用类似于其他语言的契约或接口概念所导致的，在其他语言中，这些概念用来分离类的定义和实现。

在 Python 中，隔离测试通常意味着实例化一个类或用特定的参数调用函数，然后验证结果是否和预期一致。当一个类或方法调用的一段代码不是由 Python 或标准库提供时，这个类或方法就不再是隔离的。

某些情况下，一个有用的方法是通过模拟调用来实现隔离。Mocking 意味着用模拟版本替换一段代码，接受特定的输入，产生指定的输出，并在输入和输出之间模拟行为。但 Mocking 通常具有危险，因为很容易在模拟中实现与真实对象不同的行为，最终导致一些代码可基于模拟对象工作，却无法基于真实对象工作。常发生问题的情况是，当更新项目的依赖项时，其中一些库可能引入新行为，模拟对象没有按照这些新行为进行更新。

因此，一个好的实践是，对于下面 3 种用例限制使用模拟对象：

- **I/O 操作**：当代码对第三方服务或资源(Socket、文件等)进行调用，而你无法在测试环境中运行这些资源时。
- **CPU 密集操作**：当调用计算会使测试套件太慢时。
- **要重现的特定行为**：当你要编写针对特定行为的测试代码时(比如，一个网络错误，或通过模拟日期和时间模块来改变日期或时间)。

下面这个类使用 requests 库(http://docs.python-requests.org)，通过 Bugzilla 的 REST API 来查询 bug 列表：

```
import requests
```

```
class MyBugzilla:
    def __init__(self, account, server =
                 'https://bugzilla.mozilla.org'):
        self.account = account
        self.server = server
        self.session = requests.Session()

    def bug_link(self, bug_id):
            return '%s/show_bug.cgi?id=%s' % (self.server, bug_id)

    def get_new_bugs(self):
        call = self.server + '/rest/bug'
        params = {'assigned_to': self.account,
                  'status': 'NEW',
                  'limit': 10}
        try:
            res = self.session.get(call, params=params).json()
        except requests.exceptions.ConnectionError:
            # oh well
            res = {'bugs': []}

        def _add_link(bug):
            bug['link'] = self.bug_link(bug)
            return bug

        for bug in res['bugs']:
            yield _add_link(bug)
```

这个类有一个可独立测试的 bug_link()方法。还有一个需要调用 Bugzilla 服务器的 get_new_bugs()方法。执行测试时，运行 Bugzilla 服务器过于复杂；为此，你可通过模拟这个调用并提供自定义的 JSON 数据，让这个类独立工作。

下例通过使用 request_mock(http://requests-mock.readthedocs.io)实现了模拟，这个库可轻而易举地模拟网络调用。

```
import unittest
from unittest import mock
import requests
from requests.exceptions import ConnectionError
import requests_mock
```

```
from bugzilla import MyBugzilla

class TestBugzilla(unittest.TestCase):
    def test_bug_id(self):
        zilla = MyBugzilla('tarek@mozilla.com', server =
                           ='http://example.com')
        link = zilla.bug_link(23)
        self.assertEqual(link, 'http://example.com/show_bug.cgi?id=23')

    @requests_mock.mock()
    def test_get_new_bugs(self, mocker):
        # mocking the requests call and send back two bugs
        bugs = [{'id': 1184528}, {'id': 1184524}]
        mocker.get(requests_mock.ANY, json={'bugs': bugs})

        zilla = MyBugzilla('tarek@mozilla.com',
                           server ='http://example.com')
        bugs = list(zilla.get_new_bugs())
        self.assertEqual(bugs[0]['link'],
                         'http://example.com/show_bug.cgi?id=1184528')

    @mock.patch.object(requests.Session, 'get',
                       side_effect=ConnectionError('No network'))
    def test_network_error(self, mocked):
        # faking a connection error in request if the web is down
        zilla = MyBugzilla('tarek@mozilla.com',
                           server='http://example.com')

        bugs = list(zilla.get_new_bugs())
        self.assertEqual(len(bugs), 0)

if __name__ == '__main__':
    unittest.main()
```

当项目增长时，要时刻注意这些模拟对象，对于特定功能，要确保包含模拟对象的测试不是唯一测试。例如，如果 Bugzilla 项目使用了新的 REST API 结构，且项目使用的服务器也被更新，这时只有及时根据新的 API 行为更新模拟对象，被破坏的代码才能通过测试。

test_network_error()方法是第二个测试，通过使用 Python 的 mock 补丁进行装饰，可触发请求链接错误来冒充网络错误。这个测试确保被测类能在没有网络时如期工作。

单元测试类型通常足可用于对大部分类和函数的行为进行测试。

当项目增长和新场景出现时，这个测试类将覆盖更多案例。例如，当服务器返回一个格式不正确的 JSON body 时，会发生什么？

不过，没必要从一开始就对能想出的所有失败都编写测试。在微服务项目中，单元测试不是最重要的，而且要做到 100%的单元测试覆盖率(调用测试时，运行的代码行数占所有代码行数的比例)很可能得不偿失，你需要做大量维护工作，而收益却不多。

更好的方式是集中精力构建一套健壮的功能测试。

3.1.2 功能测试

微服务项目的功能测试是：通过发送 http 请求并断言 http 响应，所有的测试与发布的 API 进行交互。

这个定义很宽泛，包含任何能调用应用的测试，从模糊测试(给应用发送没有意义的请求，看看会发生什么)乃至渗透测试(尝试破坏应用的安全)等。

开发者主要关注两类最重要的功能测试：

- 验证应用的行为和期望一致的测试
- 确保异常行为已被修复且不再发生的测试

在测试类中，开发者自行决定如何组织这些场景，但通用模式是在测试类中创建一个应用的实例，然后通过这个实例与应用进行交互。

这种场景下，没有使用网络层，并且应用也是直接被测试调用的，但发生了和使用网络层相同的请求-响应周期，因此是足够真实的。但是，仍然会模拟应用中发生的任何网络调用。

Flask 中有一个 FlaskClient 类，用来构建请求，可直接从 app 对象的 test_client()方法中获取该类的实例。

下例用于测试前面显示的第一个应用，应用会对发给/api/的请求返回 JSON body：

```
import unittest
import json
from flask_basic import app as tested_app

class TestApp(unittest.TestCase):
    def test_help(self):
        # creating a FlaskClient instance to interact with the app
```

```
        app = tested_app.test_client()

        # calling /api/ endpoint
        hello = app.get('/api')

        # asserting the body
        body = json.loads(str(hello.data, 'utf8'))
        self.assertEqual(body['Hello'], 'World!')

if __name__ == '__main__':
    unittest.main()
```

FlaskClient 类对每个 http 动词都有一个方法，方法会返回响应对象，然后这些对象可以被测试用来对结果进行验证。在上例中，我们使用.get()方法来获得响应对象。

在 Flask 类中有一个 testing 标志，可用它将异常传递到测试中，但有时倾向于不按照默认方式从应用得到返回值。比如为让 API 保持一致，要确保把返回体中的 5*xx* 或 4*xx* 错误转换成 JSON 格式。

在下例中，调用/api/会产生一个异常，通过结构化的 JSON 返回体，测试要确保客户端在 test_raise()中获取正确的 500 信息。

test_proper_404()测试方法对一个不存在的路径进行了相似的校验。

```
import unittest
import json
from flask_error import app as tested_app

_404 = ('The requested URL was not found on the server. '
        'If you entered the URL manually please check your '
        'spelling and try again.')
    class TestApp(unittest.TestCase):
        def setUp(self):
            # creating a client to interact with the app
            self.app = tested_app.test_client()

        def test_raise(self):
            # this won't raise a Python exception but return a 500
            hello = self.app.get('/api')
            body = json.loads(str(hello.data, 'utf8'))
            self.assertEqual(body['code'], 500)
```

```
        def test_proper_404(self):
            # calling a non existing endpoint
            hello = self.app.get('/dwdwqqwdwqd')

            # yeah it's not there
            self.assertEqual(hello.status_code, 404)

            # but we still get a nice JSON body
            body = json.loads(str(hello.data, 'utf8'))
            self.assertEqual(body['code'], 404)
            self.assertEqual(body['message'], '404: Not Found')
            self.assertEqual(body['description'], _404)

if __name__ == '__main__':
    unittest.main()
```



WebTest(http://webtest.pythonpaste.org)可替代 FlaskClient，它提供更多扩展的功能，本章稍后会介绍它。

3.1.3 集成测试

单元测试和功能测试主要是在不调用其他网络资源的情况下测试代码，不论其他资源是应用中的其他微服务，还是数据库、消息队列之类的第三方服务。为提高测试速度，以及让测试保持独立、简单，网络请求都被模拟了。

集成测试是不进行任何模拟的功能测试，能在应用的真实部署环境上运行。例如，如果服务与 Redis 或 RabbitMQ 有交互，在运行集成测试时，这些第三方服务也会被正常地调用。

集成测试的优势是避免了前面描述的模拟网络交互中的问题。只有在真实环境中测试应用，才能确保它将在生产环境上正常工作。

在真实部署环境进行测试要注意的是，不论准备测试数据还是清理测试期间产生的数据都是困难的。通过重现问题来修补应用的行为也是艰难的任务。

但配合单元测试和功能测试，对于验证应用的行为，集成测试是绝佳的补充。

一般而言，集成测试在服务开发期间或临时部署期间执行。但如果部署环境很容易，也可以有一个专门的测试部署环境，这个环境只用来进行集成测试。

可使用自己喜欢的任何工具来编写集成测试，例如，使用 curl 脚本来测试一些微服务有时就足够了。

不过，最好用 Python 来编写集成测试，并让其成为项目测试集的一部分。为此，可通过 Python 脚本使用请求来调用微服务，或用一种更好的方式，给微服务提供一个客户端库，并用它来进行测试。



集成测试与功能测试的主要区别在于，运行集成测试时，会调用真实的服务器。如何编写一个功能测试，既能在本地 Flask 应用上运行，又能在真实部署环境中运行呢？使用 WebTest 可解决这个问题，见稍后的介绍。

3.1.4　负载测试

负载测试的目的是了解在压力情景下服务的瓶颈，并为未来做好准备，而不是过早地进行优化。

对于服务的使用场景来说，也许第一个版本就已经足够快了，但理解它的极限将帮助你决定如何进行部署，以及如何设计才能支撑未来负载的增长。

常见的错误是花费了很多时间让每个微服务都尽可能快，但因为你的设计难以将特定的微服务部署成多个实例，所以最终应用因为只能单点部署而失败。

编写负载测试可回答下列问题：

- 将服务部署到这个机器上时，服务的一个实例能支持多少用户？
- 当有 10、100 或 1000 个并发请求时，平均响应时间是多久？服务能处理这么多并发请求吗？
- 当微服务承受压力时，是否会耗尽所有内存或 CPU？
- 能否通过为相同服务添加多个实例来进行水平扩展？
- 当我的微服务调用其他服务时，能使用连接器池吗？还是必须在一个连接内序列化地执行所有交互？
- 在不降级的情况下，服务能一次运行多天？
- 经过一次使用峰值后，服务是否还能正常工作？

基于要达到的不同负载目标，可使用很多工具，从量级较轻的命令行工具到量级较重的分布式负载系统都是备选工具。

当执行一个简单的负载测试时，不需要准备特殊的场景。Boom(https://github.com/tarekziade/boom)是一个用 Python 编写的工具，它与 Apache Bench 等效，可用来测试你的端点。

在下例中，Boom 针对路径/api/端点上的 Flask Web 服务器运行一个持续 10 秒的负载测试，模拟了 100 个并发用户，每秒请求数(Requests Per Second，RPS)是 286 个。

```
$ boom http://127.0.0.1:5000/api -c 100 -d 10 -q
```

```
-------- Results --------
Successful calls          2866
Total time                10.0177 s
Average                   0.3327 s
Fastest                   0.2038 s
Slowest                   0.4782 s
Amplitude                 0.2744 s
Standard deviation        0.035476
RPS                       286
BSI                       Pretty good

-------- Status codes --------
Code 200                  2866 times.

-------- Legend --------
RPS: Request Per Second
BSI: Boom Speed Index
```

有些负载测试的结果数据并不是很有意义，因为它们依赖于部署方案，也依赖于运行环境。例如，如果你的 Flask 应用部署在 nginx 服务器后面，而且一个应用有多个实例，将能更好地处理传入的连接流。

但这个小测试通常可在早期发现问题，特别是当代码正在打开 Socket 连接时。若微服务结构设计出现错误，那么通过使用诸如 Boom 的工具，很容易就能发现问题。

当需要编写有交互场景的负载测试时，另一个小巧的命令行工具是 Molotov(https://github.com/tarekziade/molotov)。你可使用该工具编写一个 Python 函数，用这个函数来查询微服务并校验服务的响应。

下例中，通过被 Molotov 选取后在服务器上运行，每个函数都代表一个可能的场景：

```
import json
from molotov import scenario

@scenario(5)
async def scenario_one(session):
    res = await session.get('http://localhost:5000/api').json()
    assert res['Hello'] == 'World!'
@scenario(30)
async def scenario_two(session):
```

```
somedata = json.dumps({'OK': 1})
res = await session.post('http://localhost:5000/api',
                         data=somedata)
assert res.status_code == 200
```

这两个工具都可提供一些度量指标，但并不是精准，因为运行测试的网络和客户端 CPU 存在差异。例如，如果测试本身就耗用硬件资源，将影响测试指标。

当执行负载测试时，最好在服务器端添加一些度量值。在 Flask 级别，可使用诸如 Flask Profiler(https://github.com/muatik/flask-profiler)的小工具，它可用来收集每个请求花了多长时间，并提供一个仪表盘来显示收集时间(开销极小)，如图 3-2 所示。



还可通过 StatsD(https://github.com/etsy/statsd)来发送详细指标，并使用诸如 Graphite(http://graphite.readthedocs.io)的专用仪表盘应用。第 6 章将再次提到相关的度量。

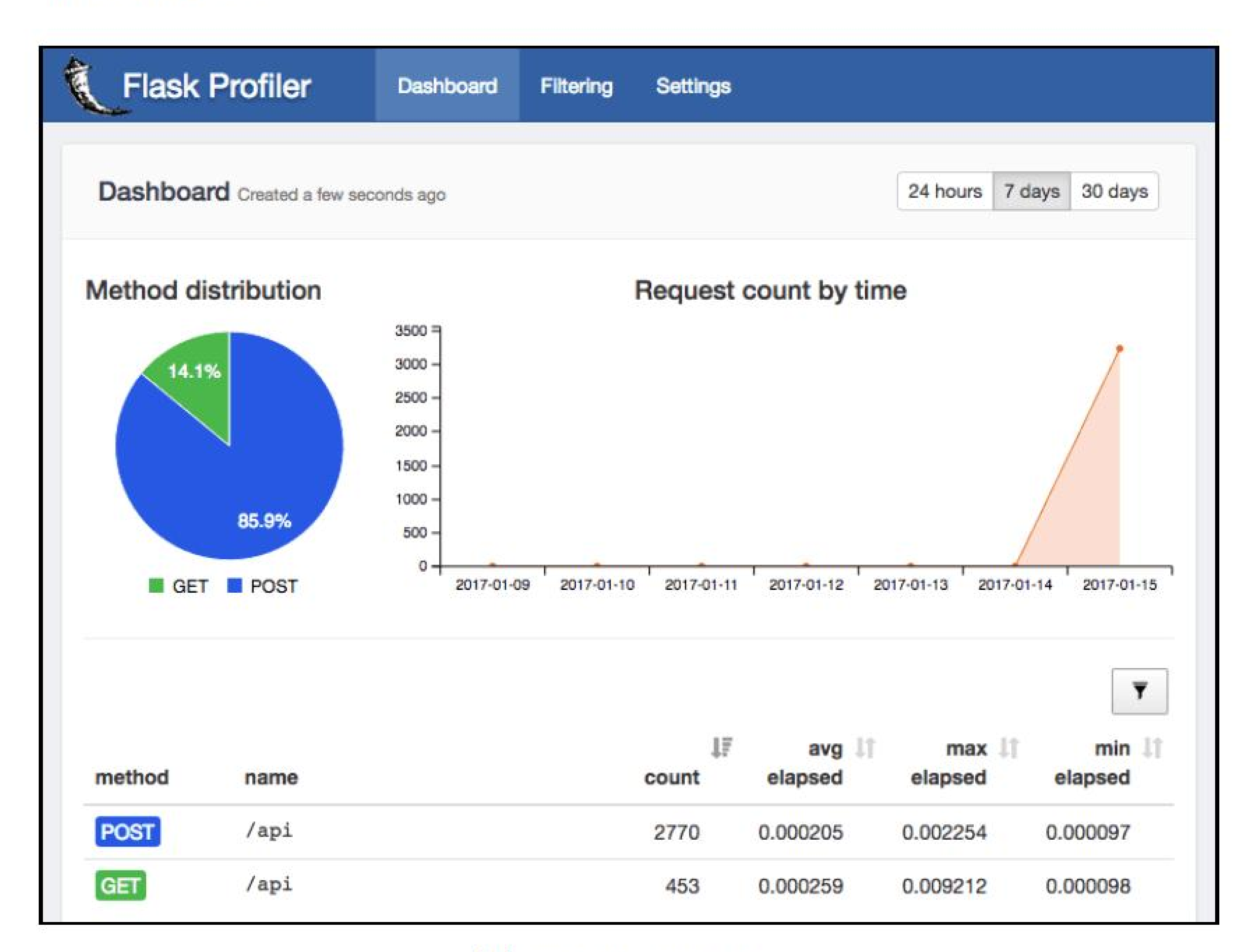


图 3-2　Flask Profiler

如果想要执行量级更重的负载测试，则需要使用一些负载测试框架，将测试分发给多个代理。一个备选工具是 locust.io(http://docs.locust.io/)。

3.1.5　端到端测试

端到端测试从终端用户的视角来检查整个系统是否符合期望。测试需要像真实客

户的行为，通过相同的 UI 来调用系统。

根据所编写应用的不同类型，一个简单的 HTTP 客户端可能不足以模拟一个真实用户。例如，如果用户正在交互的系统的可视化部分是一个包含 HTML 网页的 Web 应用，HTML 网页的渲染是在客户端完成的，将需要使用诸如 Selenium (http://docs.seleniumhq.org/)的工具来模拟用户交互。它可自动操作浏览器，来确保客户端请求每个 CSS 和 JavaScript 文件，并正确地渲染每个页面。

JavaScript 框架在客户端完成许多工作来生成页面。一些已完全移除了服务端渲染模板，只是从服务端获取数据，然后通过浏览器 API 来操作 DOM(Document Object Model)，用获取的数据生成 HTML 页面。这种情况下，当通过指定 URL 请求服务器时，返回的结果由用来渲染的所有静态 JavaScript 文件和数据组成。



如何编写端到端测试不属于本书的讨论范围，如果需要了解相关信息，可参阅 *Selenium Testing Tools Cookbook*。

下面总结本节讨论的要点：

- 功能测试是要编写的最重要测试，通过在测试中实例化一个 app 并与其交互，可很容易地在 Flask 中编写功能测试。
- 单元测试是功能测试的良好补充，但不要滥用模拟方法。
- 集成测试类似于功能测试，但在真实发布环境中运行测试。
- 负载测试有助于了解微服务的瓶颈，以便制定下一步改进计划。
- 端到端测试要求使用客户惯用的 UI 进行测试。

何时编写集成测试、负载测试和端到端测试？这要依据项目的管理方式而定(但是，每次执行改动时，都应编写单元测试和功能测试)。每次修改代码时，理想情况下都应创建一个新测试或修改一个旧测试。

由于标准库中已经包含好用的 unittest 包，因此可使用普通 Python 方式来编写单元测试(稍后将看到如何使用 pytest，在此基础之上添加更卓越的功能)。

下一节将介绍用于功能测试的 WebTest。

3.2 使用 WebTest

WebTest(http://webtest.readthedocs.io)已经存在很长时间了，它是 Ian Bicking 在开发 Paste 项目时编写的。它基于 WebOb(http://docs.webob.org)项目，WebOb 提供的功能类似于 Flask 中的 Request 和 Response 类(但不兼容)。

与 FlaskTest 一样，WebTest 也包装了对 WSGI 应用的调用，并基于这种方式进行

交互。WebTest 在某些方面类似于 FlaskTest，如处理 JSON 时不需要额外帮助，而且 WebTest 还能简洁地调用非 WSGI 应用。

为配合 Flask 使用，可安装 flask-webtest 包(https://flask-webtest.readthedocs.io/)，然后即可像使用 Flask 原生工具一样使用它：

```
import unittest
from flask_basic import app as tested_app
from flask_webtest import TestApp

class TestMyApp(unittest.TestCase):
    def test_help(self):
        # creating a client to interact with the app
        app = TestApp(tested_app)

        # calling /api/ endpoint
        hello = app.get('/api')

        # asserting the body
        self.assertEqual(hello.json['Hello'], 'World!')
if __name__ == '__main__':
    unittest.main()
```

之前提到过集成测试类似于功能测试，但集成测试会请求真实的服务器，而非本地的 WSGI 应用。

WebTest 利用了 WSGIProxy2 库(https://pypi.python.org/pypi/WSGIProxy2)，这个库可将对 Python 应用的调用转换成对实际 HTTP 应用的 HTTP 请求。

只需要在 environ 函数中设置 HTTP_SERVER 变量，即可轻松地将之前的功能测试代码改成集成测试：

```
import unittest
import os
class TestMyApp(unittest.TestCase):

    def setUp(self):
        # if HTPP_SERVER is set, we use it as an endpoint
        http_server = os.environ.get('HTTP_SERVER')
        if http_server is not None:
            from webtest import TestApp
            self.app = TestApp(http_server)
```

```
            else:
                # fallbacks to the wsgi app
                from flask_basic import app
                from flask_webtest import TestApp
                self.app = TestApp(app)

        def test_help(self):
            # calling /api/ endpoint
            hello = self.app.get('/api')

            # asserting the body
            self.assertEqual(hello.json['Hello'], 'World!')

    if __name__ == '__main__':
        unittest.main()
```

通过设置 HTTP_SERVER=http://myservice/执行最后一个测试时，所有调用都会指向 myservice。

这个小技巧可方便地将功能测试转变成集成测试，而不必写两套测试。之前提到这种方式也有一些限制，如不能与本地应用实例进行交互。但若想从测试套件验证部署的服务能否正常工作，这个技巧将非常有用，因为只需要修改一个选项。

3.3 使用 pytest 和 Tox

我们迄今为止编写的所有测试都使用 unittest.TestCase 类和 unittest.main()来运行。当项目增长时，会添加越来越多的测试模块。

为自动发现和运行一个项目的所有测试，Python 3.2 的 unittest 包引入了测试发现功能，此功能可根据给定的选项查找测试，然后运行它们。在 Nose 和 pytest 等项目中已使用类似的功能很久了，这催生了 unittest 标准库。

使用什么运行器是个人偏好，只要坚持在 TestCase 类中编写测试，测试将与所有测试运行器兼容。

pytest 项目在 Python 社区非常流行。并且因为它支持扩展，大家能基于它来编写好用的工具。它的测试运行器也十分高效，后台发现测试时就可以开始运行测试了，这使得它比其他运行器更快。控制台的输出也很美观。

要在项目中使用它，用 pip 命令可很容易完成安装，然后即可在命令行中使用提供的 pytest 命令了。下例中，pytest 命令会运行所有以 test_开头的模块：

```
$ pytest test_*
============= test session starts ================================
platform darwin -- Python 3.5.2, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/tarek/Dev/github.com/microbook/code, inifile:
collected 7 items

test_app.py .
test_app_webtest.py .
test_bugzilla.py ...
test_error.py ..

============= 7 passed in 0.55 seconds ============================
```

在http://plugincompat.herokuapp.com/页面能找到pytest包附带的大量扩展。pytest-cov和pytest-flake8 是两个有用的扩展。前一个使用coverage工具(https://coverage.readthedocs.io)来显示项目的测试覆盖率，后一个运行Flake8(https://gitlab.com/pycqa/flake8)linter来确保代码遵循PEP8 标准，而且不存在未使用的导入。

下面是一个调用示例，其中一些样式问题是故意留下的：

```
$ pytest --cov=flask_basic --flake8 test_*
============= test session starts ================================
platform darwin -- Python 3.5.2, pytest-3.0.5, py-1.4.32, pluggy-0.4.0
rootdir: /Users/tarek/Dev/github.com/microbook/code, inifile:
plugins: flake8-0.8.1, cov-2.4.0
collected 11 items

test_app.py F.
test_app_webtest.py F.
test_bugzilla.py F...
---------- coverage: platform darwin, python 3.5.2-final-0 -----------
Name Stmts Miss Cover
--------------------------------------
flask_basic.py 6 1 83%
============= FAILURES ======================================
_______________ FLAKE8-check ____________________________________
test_app.py:18:1: E305 expected 2 blank lines after class or function
definition, found 1
test_app.py:21:1: W391 blank line at end of file
_______________ FLAKE8-check ____________________________________
```

```
test_app_webtest.py:29:1: W391 blank line at end of file
_______________ FLAKE8-check ____________________________________
test_bugzilla.py:26:80: E501 line too long (80 > 79 characters)
test_bugzilla.py:28:80: E501 line too long (82 > 79 characters)
test_bugzilla.py:40:1: W391 blank line at end of file

============= 3 failed, 7 passed, 0 skipped in 2.19 seconds =============
```

Tox(http://tox.readthedocs.io)是另一个与 pytest 结合使用的工具。

如果需要在多个 Python 版本上运行项目，或者只想确保代码在最新的 Python 2 和 Python 3 版本上都可工作，Tox 能自动创建独立的环境来运行测试。

通过安装 Tox(使用 pip install tox 命令)，然后在项目中创建一个 tox.ini 配置文件，就能告诉它将在 Python 2.7 和 Python 3.5 上运行项目。Tox 会假设你的项目是一个 Python 包，在包的根目录下有一个 setup.py 文件，使用 Tox 的唯一要求是将 tox.ini 和 setup.py 放在一起。

tox.ini 文件包运行测试的命令行，命令行会根据设定的 Python 版本来执行：

```
[tox]
envlist = py27,py35

[testenv]
deps = pytest
    pytest-cov
    pytest-flake8

commands = pytest --cov=flask_basic --flake8 test_*
```

当调用 tox 命令执行 Tox 时，对于每个 Python 版本，都会创建独立的测试环境，然后部署包和相关依赖项，最后在测试环境中使用 pytest 运行测试。

当希望更快运行测试时，一个有效方法是使用 tox -e 运行独立的环境。例如，tox -e py35 仅在 Python 3.5 环境下运行 pytest。

即便只需要支持一个 Python 版本，只要已经正确地定义了所有依赖项，使用 Tox 将确保项目被安装到当前的 Python 环境中。

强烈推荐使用这个工具。



第 9 章将详细讲解如何打包微服务，将使用 Tox 和其他工具一起完成打包。

3.4　开发者文档

至此，已经介绍了针对微服务的各种测试，前面提到过文档应该和代码一起演进。

现在开始介绍开发者文档，它包含开发者应该知道的有关微服务项目的一切，比如：

- 设计方式
- 安装方式
- 如何运行测试
- 公开的 API 有哪些，流入流出的数据都有哪些，等等

Sphinx 工具已成为 Python 社区的标准，用于为 Python 编写文档。

通过将内容和布局分开，Sphinx 像源代码一样处理文档。使用 Sphinx 的常见方法是在项目中布置一个 docs 目录，用来存放文档内容，然后调用 Sphinx 的命令行工具来生成文档，生成的文档会使用一种输出格式，如 HTML 格式。

使用 Sphinx 生成的 HTML 输出结果能直接生成优秀的静态网站，静态网站可直接发布到网上，其中包含索引页、一个基于 JavaScript 的小搜索引擎以及导航功能。

必须使用 reStructuredText(http://docutils.sourceforge.net/rst.html)语言来编写文档内容，这是 Python 社区中一个常用的标准标记语言。reST 文件是一个简单文本文件，它具有非侵入性的语法特点，能标记章节标题、链接、文本样式等。Sphinx 添加了一些扩展，并在文档中总结了如何使用 reST，通过访问 http://www.sphinx-doc.org/en/latest/rest.html 可学习如何编写文档。



Markdown(https://daringfireball.net/projects/markdown/)是另一种流行于开源社区的标记语言。遗憾的是，由于 Sphinx 依赖一些 reST 扩展，通过 recommonmark 包只能部分支持 Markdown。但好消息是，如果你熟悉 Markdown，reST 也没有太大区别。

通过 sphinx-quickstart 命令在项目中使用 Sphinx 时，它会在 index.rst 文件中生成源码树，这是文档的入口页面。然后使用这个文件调用 sphinx-build 命令来创建文档。

例如，要生成 HTML 文档，可在 tox.ini 文件中添加一个 docs 环境变量，让工具自动构建文档，像下面这样：

```
[tox]
envlist = py35,docs
...
```

```
[testenv:docs]
basepython=python
deps =
    -rrequirements.txt
    sphinx
commands=
    sphinx-build -W -b html docs/source docs/build
```

然后运行 tox -e docs 就能生成文档了。

通过将代码粘贴到一段文本块中，然后在文本块前标记::前缀和 code-block 指令，就能在Sphinx 中显示代码示例了。在HTML 中，Sphinx 将使用 Pygments(http://pygments.org/)语法高亮渲染代码：

```
Flask Application
=============

Below is the first example of a **Flask** app in the Flask official doc:

.. code-block:: python

    from flask import Flask
    app = Flask(__name__)

    @app.route("/")
    def hello():
        return "Hello World!"
    if __name__ == "__main__":
        app.run()

That snippet is a fully working app!
```

但一旦代码发生改变，文档中添加的代码段将随之过时。为避免过时，一种方法是从源代码抽取文档中显示的代码片段。

为此，可使用 docstring 在源代码的模块、类或函数上编写文档，然后使用 Autodoc Sphinx 扩展(http://www.sphinx-doc.org/en/latest/ext/autodoc.html)，这个扩展可从源代码中提取 docstring，然后插入文档中。

要查看Python是如何文档化其标准库的，可访问页面https://docs.python.org/3/library/index.html。下例中，autofunction指令会从myservice/views/home.py模块的index函数中抽取docstring来生成文档。

```
APIS
====

**myservice** includes one view that's linked to the root path:

.. autofunction :: myservice.views.home.index
```

如果渲染成 HTML 页面，将如图 3-3 所示：

APIS

myservice includes one view that's linked to the root path:

myservice.views.home.**index**()

Home view.

This view will return an empty JSON mapping.

©2017, joe. | Powered by Sphinx 1.5.1 & Alabaster 0.7.9 | Page source

图 3-3　抽取 docstring 生成 HTML 文档

另一个选项是使用 literalinclude 指令，它允许指向一个文件，并提供选项来高亮显示文件的章节。若所指文件是一个 Python 模块，这个文件还会被测试套件包含以确保能正常工作。

下面是一个使用 Sphinx 编写项目文档的完整示例：

```
Myservice
=========


**myservice** is a simple JSON Flask application that uses **Flakon**.

The application is created with :func:`flakon.create_app`:
.. literalinclude:: ../../myservice/app.py


The :file:`settings.ini` file which is passed to :func:`create_app`
contains options for running the Flask app, like the DEBUG flag:
.. literalinclude:: ../../myservice/settings.ini
   :language: ini
```

```
Blueprint are imported from :mod:`myservice.views` and one
Blueprint and view example was provided in :file:`myservice/views/home.py`:

.. literalinclude:: ../../myservice/views/home.py
   :name: home.py
   :emphasize-lines: 13


Views can return simple mappings (as highlighted in the example above),
in that case, they will be converted into a JSON response.
```

页面会渲染成 HTML，如图 3-4 所示：

Myservice

myservice is a simple JSON Flask application that uses **Flakon**.

The application is created with **flakon.create_app()**:

```
import os
from flakon import create_app
from myservice.views import blueprints


_HERE = os.path.dirname(__file__)
_SETTINGS = os.path.join(_HERE, 'settings.ini')

app = create_app(blueprints=blueprints, settings=_SETTINGS)
```

The settings.ini file which is passed to **create_app()** contains options for running the Flask app, like the DEBUG flag:

```
[flask]
DEBUG = true
```

Blueprint are imported from **myservice.views** and one Blueprint and view example was provided in myservice/views/home.py:

```
from flask import Blueprint


home = Blueprint('home', __name__)


@home.route('/')
def index():
    """Home view.

    This view will return an empty JSON mapping.
    """
    return {}
```

Views can return simple mappings (as highlighted in the example above), in tha case they will be converted into a JSON response.

©2017, joe. | Powered by Sphinx 1.5.1 & Alabaster 0.7.9 | Page source

图 3-4　抽取源代码生成 HTML 文档

当然，使用 Autodoc 和 literalinclude 不会修复文档的行文或结构——要保持适当的文档同步是困难的，还需要做更多工作。

因此，任何可自动完成部分文档的工具都算是很棒的。



在第 4 章将介绍如何用 Swagger 和 sphinx-swagger 扩展为微服务 HTTP API 编写文档。

本节有四个要点：

- Sphinx 是一个用来编写项目文档的强大工具。
- 将文档当作源代码来处理，会让维护文档更加方便。
- 当代码改变时，可用 Tox 重建文档。
- 如果让文档指向代码，维护文档将更容易。

3.5　持续集成

当项目发生改变时，Tox 可自动执行下面的每一步：在各种 Python 解释器上运行测试，验证测试覆盖率，检查是否满足 PEP8 的要求，构建文档，等等。

但如果每次改变都运行所有检查，会耗费不少时间和资源，尤其当项目需要支持多个解释器时。

持续集成(Continue Integration)系统可解决这个问题，每次项目发生改变时，CI 会接管这些工作。

将项目推送到一个基于 DVCS (Distributed Version Control System，分布式版本控制系统)的共享代码库，例如 Git 或 Mercurial。每次有人将代码推入代码库服务器后，代码库服务器可触发一个 CI。

如果只在一个开源软件上工作，而且并不打算维护自有的代码服务器，最流行的代码服务器有 GitHub(http://github.com)、GitLab(http://gitlab.com)和 BitBucket(https://bitbucket.org/)。如果是公开项目，它们会免费维护，并提供社交功能，社交功能可让其他人很容易地为项目贡献代码。当项目发生变更时，这些代码服务器都提供了集成点用来运行任何需要运行的脚本。

例如，在 GitHub 上，如果在 reST 文档中看到一个拼写错误，可直接从浏览器修改它，预览结果，然后通过点击按钮，就能向项目维护者发送一个 PR(Pull Request)。此后项目会自动开始重建，一旦完成，构建状态将显示在 PR 上。

许多开源项目使用这些服务创建一个有很多贡献者的繁荣社区。Mozilla 在 Rust 项目上使用了 GitHub，毫无疑问，这有助于吸引更多贡献者参与进来。

3.5.1 Travis-CI

GitHub 可直接与一些 CI 集成，一个非常流行的 CI 是 Travis-CI(https://travis-ci.org/)，开源项目可免费运行它。一旦拥有 Travis-CI 账户，就可在设置页面上激活存放在 GitHub 上的项目。

Travis-CI 依赖于.travis.yml YAML 文件，该文件需要放在代码库的根目录下，它描述了当项目发生改变时需要做什么。

这个 YAML 文件包含一个 env 块，用来描述构建的 matrix。matrix 是一组构建的集合，每次项目发生改变时，将可并行地运行。

matrix 可与 Tox 环境相匹配，通过 tox -e 可独立地在每个环境中运行。这样，你能知道何时由于更改破坏了一个特定环境。

```
language: python
python: 3.5
env:
 - TOX_ENV=py27
 - TOX_ENV=py35
 - TOX_ENV=docs
 - TOX_ENV=flake8
install:
 - pip install tox
script:
 - tox -e $TOX_ENV
```

Travis-CI 包含与 Python 项目一起工作需要的一切，所以可在 install 部分用 pip 命令安装 Tox，然后启动构建。



tox-travis 是一个有趣的项目，它扩展了 Tox 来简化 Travis 集成。它提供的功能像一个环境检测器，简化了 tox.ini 文件的编写。

如果有系统级依赖项，可通过 YAML 文件进行安装，甚至运行 bash 命令。默认环境运行 Linux Debian，你可直接在 YAML 文件的 before-install 部分键入 apt-get 命令。

Travis 也支持设置特殊服务(参考 https://docs.travis-ci.com/user/database-setup/)，如数据库等。通过对 services 部分进行配置，就能为项目部署特殊服务。

如果微服务使用 PosgtreSQL、MySQL 或其他主流开源数据库，它们可能是直接可用的。如果没有使用，可直接编译数据库并在构建上运行它。当使用 Travis-CI 时，Travis 文档(https://docs.travis-ci.com/)是一个提供帮助的好地方。



Travis 可在 Linux 代理上触发构建，也能有限地支持 macOS X。遗憾的是，还不支持 Windows。

3.5.2 ReadTheDocs

与Travis使用方法一样，另一个可挂在GitHub代码库上的服务是RTD(ReadTheDocs，https://docs. readthedocs.io)。

不需要在代码库中做任何事情，它就能生成项目文档并托管文档。只需要配置 RTD，即可从 Sphinx HtmlDir 创建文档，服务会自动找到相关元素。

对于非 Travis 集成，RTD 可通过 YAML 文件进行配置。一旦文档准备就绪，就可通过 https://<yourprojectname>.readthedocs.io 进行访问。

RTD 还附带版本支持功能，当发布服务的新版本时非常有用。该功能会扫描 Git 标签，根据每个标签构建和发布文档，然后决定哪一个是默认的。

与版本功能一样，如果需要使用多国语言编写文档，RTD 还提供国际化(i18n)支持。

3.5.3 Coveralls

当 CI 使用 Travis-CI，代码库使用 GitHub 或 Bitbucket 时，另一个可挂在代码库上的常用服务是 Coveralls。这个服务能通过美观的 Web UI 显示测试覆盖率。

一旦在 Coveralls 账号下添加了代码库，通过在测试完成后指示 Tox 连接到域名 http://coveralls.io，即可直接从 Travis-CI 触发对 http://coveralls.io 的调用。

下面的[testenv]块已完成了对 tox.ini 文件的修改，用粗体显示：

```
[testenv]
passenv = TRAVIS TRAVIS_JOB_ID TRAVIS_BRANCH
deps = pytest
    pytest-cov
    coveralls
    -rrequirements.txt

commands =.
    pytest --cov-config .coveragerc --cov myservice myservice/tests
- coveralls
```

pytest 调用完成后，coveralls-python 包(在 PyPI 中名为 Coveralls)用于通过 coveralls 命令将负载发送给 coveralls.io。

注意调用的前缀是短划线(-)。像 Makefiles 中一样，这个前缀会忽略任何失败，在

本地运行 Tox 时会阻止 Tox 失败。除非设置一个包含身份验证令牌的特殊.coveralls.yml 文件，否则在本地运行 Coveralls 总是会失败。当从 Travis-CI 运行 Coveralls 时，并不需要这个设置，这要感谢从 GitHub 传给其他服务的 Token 的魔力。

要从 Travis 使用 Coveralls，需要通过 passenv 传递多个环境变量；其他的会自动运行。

项目和 Travis-CI 上的任何改变都会触发构建，构建反过来会触发 Coveralls 概括性显示测试覆盖率及其随着时间发生的变化，如图 3-5 所示。

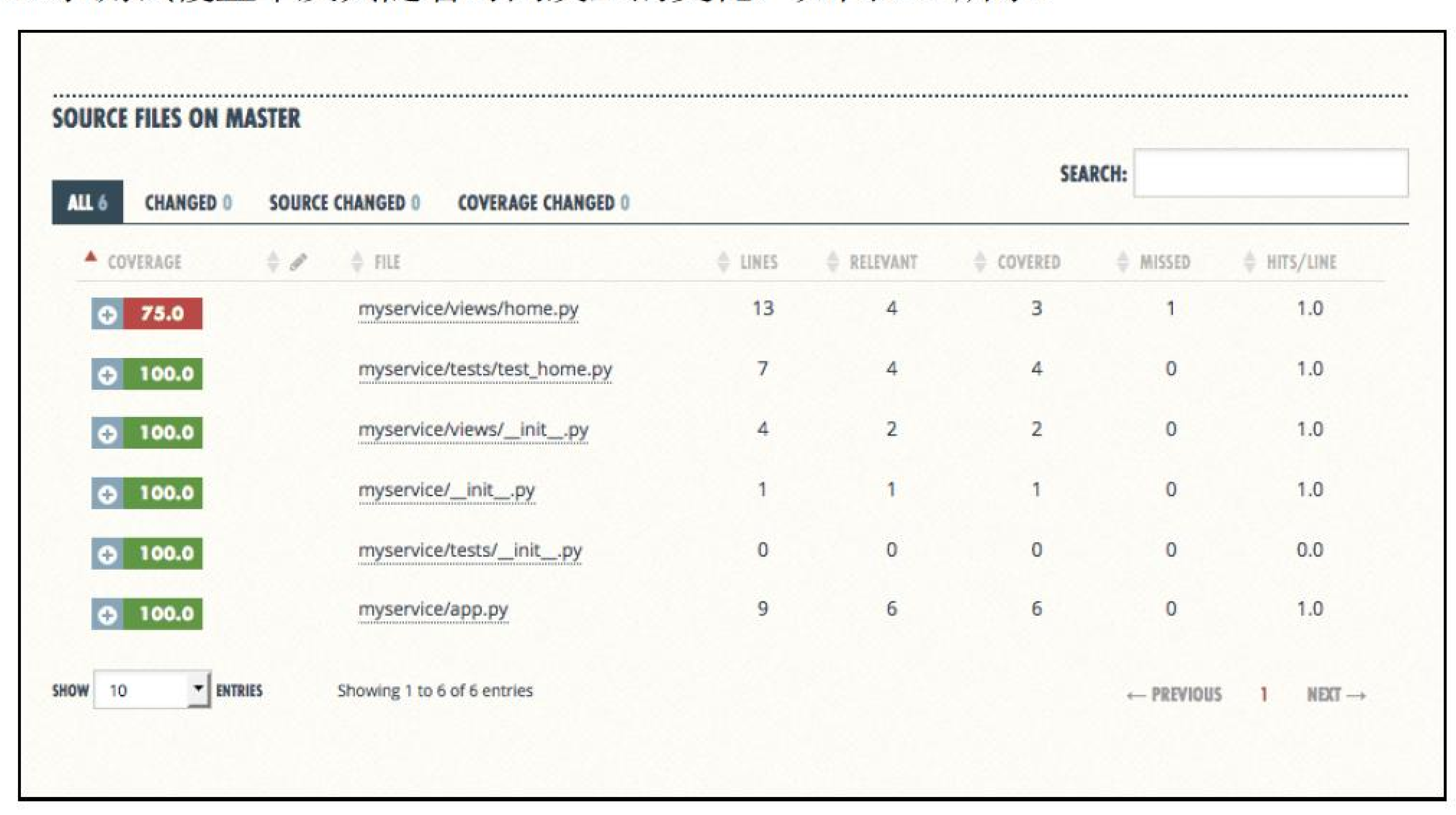


图 3-5　Coveralls 页面

还有很多服务可挂在 GitHub 或 Travis-CI 上，Coveralls 只是其中的一个例子。

一旦开始给项目添加服务，最好在项目的 README 中使用对应的标志，这样通过服务链接，社区人员一眼即可看到每个服务的状态。

例如，在代码库中添加 README.rst 文件：

```
microservice
============

This project is a template for building microservices with Flask.

.. image::
https://coveralls.io/repos/github/tarekziade/microservice/badge.
svg?branch=master
   :target:
https://coveralls.io/github/tarekziade/microservice?branch=master
```

```
.. image:: https://travis-ci.org/tarekziade/microservice.svg?branch=master
   :target: https://travis-ci.org/tarekziade/microservice

.. image::
https://readthedocs.org/projects/microservice/badge/?version=latest
   :target: https://microservice.readthedocs.io
```

在 GitHub 的项目首页上，上述文件的显示效果如图 3-6 所示。

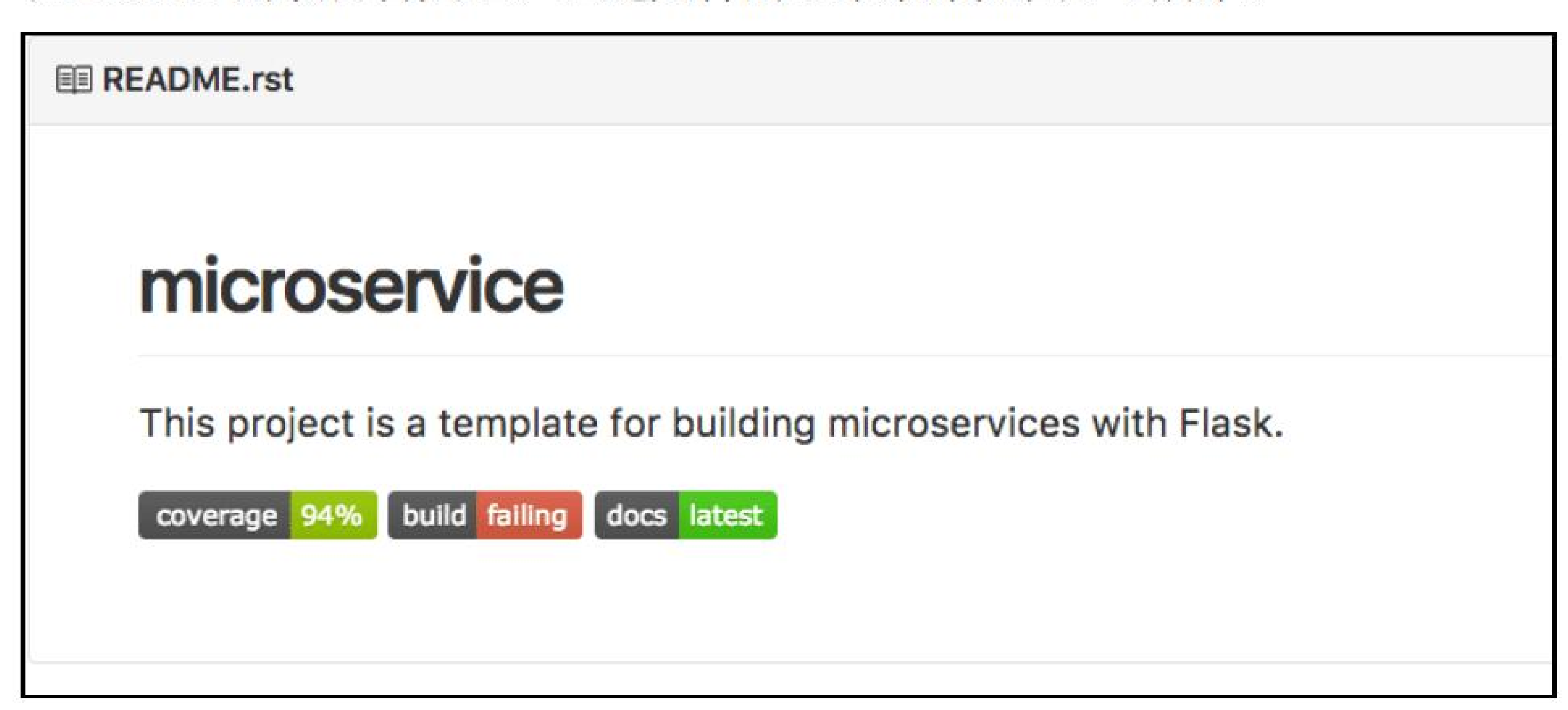


图 3-6　README 页面

3.6　本章小结

本章首先介绍微服务项目的各种测试类型。功能测试是最常编写的类型，而 WebTest 是一个非常好用的工具。pytest 配合 Tox 可更轻松地运行测试。

最后要指出的是，如果将项目存放在 GitHub 上，可免费地构建整个持续集成系统，这要感谢 Travis-CI。这样，有大量免费的服务可配合 Travis 使用，如 Coveralls。还可自动地构建文档，然后发布到 ReadTheDocs 上。



为演示这一切是如何结合在一起的，GitHub 上有一个微服务项目使用了 Travis-CI、RTD 和 coveralls.io，项目发布在 https://github.com/Runnerly/microservice。

现在已经介绍了一个 Flask 项目如何做到持续地开发、测试和写文档，你可通过这些了解如何设计一个基于微服务的完整项目。下一章将介绍微服务应用的设计。

第4章 设计Runnerly

在第 1 章中提到，在构建基于微服务的应用时，最自然的方式是从一个实现了所有功能的单体版本开始，此后将其拆分成有意义的微服务。在开发的第一天就试图设计一个基于若干个微服务的架构将后患无穷。理解应用将如何组织，以及它在成熟时将如何演进，是非常困难的。

本章将构建一个实现了所有必需功能的单体应用来完成这个过程，然后考虑如何将应用分解为较小的服务，最后将以一个基于微服务的设计收尾。

本章分为以下 3 个主要部分：

- 介绍 Runnerly 应用及其用户故事
- 如何将 Runnerly 构建为一个单体应用
- 单体应用如何演进成微服务

当然在实际中，只要单体应用的设计变得成熟一些，拆分过程就会随着时间的推移而发生。但本章假设应用的第一个版本已经用了一段时间，并展现出一些正确地拆分应用的线索。

4.1 Runnerly 应用

Runnerly 是为本书创建的，是供跑步者使用的玩具应用。请不要在 App Store 或者 Play Store 搜索它，因为它并不对真实用户发布或部署。

不过，这个应用确实可以运行。可访问 GitHub 上 的 Runnerly 组织(https://github.com/Runnerly)来查找和学习它的不同组件。

Runnerly 提供一个 Web 页面，用户一眼就能看到自己的跑步活动、竞赛和训练

计划。这是一个响应式页面，因此在手机和桌面浏览器上都能展示这个应用。Runnerly也发送用户活动的月度报告。

注册到Runnerly的用户，需要将自己的账户关联到Strava上(https://www.strava.com)。此时会使用标准的OAuth2(https://oauth.net/2/)机制。



OAuth2 标准基于这样的想法：利用用户的唯一访问令牌，授权第三方应用访问服务。这个令牌是服务生成的，通常包含权限范围。Strava 有一个完整的 API 集合，它们都可用这种方式来访问，见文档 https://strava.github.io/api/v3/。

用户授权后，Runnerly 会从 Strava 中拉取跑步活动，并存入数据库。这个流程简化了许多集成工作，让应用可与大多数跑步设备兼容。如果你的设备能与 Strava 一起工作，那么它也能与 Runnerly 一起工作。

一旦数据库可从 Strava 中获取内容，应用的仪表盘(Dashboard)上就会显示最近 10 次的跑步记录，用户还可使用 Runnerly 的额外特性：竞赛、训练计划和月度报告。

现在开始通过用户故事来深入讲解 Runnerly 的特性。

用户故事

描述应用的最佳方式是使用"用户故事(user story)"。用户故事简单描述用户与应用进行的所有交互活动，通常是项目启动时最先编写的高级文档。

这些交互的细节起初十分简单，此后每次出现新的特定情况时，都会重新改进。用户故事也在检测是否要拆分微服务时特别有用：一个独立的用户故事可能就是一个微服务。

对 Runnerly 来说，可从下面的用户故事集合开始：

- 作为一个用户，我能使用邮箱来创建账户，并通过收件箱中的一个确认链接来激活账户。
- 作为一个用户，我可通过接入 Runnerly 把我的个人信息与我的 Strava 账户关联。
- 作为一个关联用户，我可在仪表盘看到最近 10 次的跑步记录。
- 作为一个关联用户，我可添加自己想参加的竞赛。其他用户也可在他们的仪表盘上看到这个竞赛信息。
- 作为一个注册用户，我会通过电子邮件收到月度报告，其中描述了我的当前活动。

- 作为一个关联用户，我可选择我打算参加的某个竞赛的训练计划，并在仪表盘上看到训练日程。其中，训练计划简单列出尚未完成的跑步活动。

上面的用户故事中，已经涌现出一些组件。不按特定的顺序，它们是：

- 应用需要一个注册机制，将用户添加到数据库，并确保用户拥有用于注册的邮箱地址。
- 应用需要使用密码对用户进行身份验证。
- 为从 Strava 中拉取数据，需要在用户档案中保存 Strava 用户令牌，这个令牌还用来调用 Strava 的服务。
- 除了跑步活动，还要在数据库中保存竞赛和训练计划。
- 训练计划是一个特定日期的跑步活动列表，以便在给定的竞赛中尽量提高成绩。创建一个训练计划，需要用户的一些信息，如年龄、性别、体重和健身级别。
- 月度报告是通过查询数据库生成的汇总信息，通过邮件发送。

有了上面的信息，就可以进行开发了。下面描述如何进行设计和编码。

4.2　单体设计

本节介绍从单体版Runnerly中提取的内容。如果想仔细地学习它，可在https://github.com/Runnerly/monolith查看完整代码。

在构建应用时，通常使用的设计模式是 MVC(Model-View-Controller，视图-模型-控制器)模式，它将代码分成以下 3 个部分：

- **模型**：用来管理数据。
- **视图**：在特定上下文中(Web 页面、PDF 页面等)展示模型。
- **控制器**：处理模型并改变它的状态。

显然，SQLAlchemy 可作为模型部分，但在 Flask 中，视图和控制器的差别较为模糊，这是因为 Flask 中的视图是一个接收请求并返回响应的函数。这个函数可展示数据，也可处理数据。所以可同时充当视图和控制器。

Django 项目将这个模式称为 MVT (Model-View-Template，模型-视图-模板)模式。视图是 Python 的可调用代码；模板是一个模板引擎，或者基于给定数据生成特定响应格式的任何东西。

例如，在 JSON 视图中，json.dumps()就是模板。当使用 Jinja 渲染 HTML 时，模板就是通过 render_template()函数调用的 HTML 模板。

任何情况下，设计应用的第一个步骤是定义模型。

4.2.1 模型

在基于 SQLAlchemy 的 Flask 应用中，模型通过类来描述，这个类也描述了数据库模式。

Runnerly 中的数据库表包括：

- **用户(User)**：包含每个用户的信息，包括用户凭证。
- **跑步(Run)**：跑步活动列表，包含从 Strava 获取的所有信息，以及训练计划中的跑步活动。
- **竞赛(Race)**：用户添加的竞赛列表，包含日期、位置和距离。
- **计划(Plan)**：训练计划，代表将要完成的跑步活动的列表。

使用flask_sqlalchemy(http://flask-sqlalchemy.pocoo.org/)这个扩展，可将Model 作为基类，来指定模型的数据库表。下面是使用SQLAlchemy来定义User表的方式：

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()

class User(db.Model):
    __tablename__ = 'user'
    id = db.Column(db.Integer, primary_key=True, autoincrement=True)
    email = db.Column(db.Unicode(128), nullable=False)
    firstname = db.Column(db.Unicode(128))
    lastname = db.Column(db.Unicode(128))
    password = db.Column(db.Unicode(128))
    strava_token = db.Column(db.String(128))
    age = db.Column(db.Integer)
    weight = db.Column(db.Numeric(4, 1))
    max_hr = db.Column(db.Integer)
    rest_hr = db.Column(db.Integer)
    vo2max = db.Column(db.Numeric(4, 2))
```

flask_sqlalchemy 会将对 SQLAlchemy 的所有调用封装起来，并在视图中公开一个数据库会话对象，用来操纵模型。

4.2.2 视图与模板

应用接收到请求时，会调用视图，flask_sqlalchemy 会在应用上下文中创建数据库

会话对象。下面是一个完整的 Flask 应用，在/users 对应的视图中，查询了上文定义的数据库模式：

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/users')
def users():
    users = db.session.query(User)
    return render_template("users.html", users=users)

if __name__ == '__main__':
    db.init_app(app)
    db.create_all(app=app)
    app.run()
```

当调用 db.session.query()方法时，会查询数据库，并将从 User 表中查询到的结果转换成 User 对象。这些对象会传给 Jinja 格式的模板 users.html，最终渲染成 HTML 页面。

在上例中，Jinja 用来生成 HTML 页面，显示用户信息，模板如下所示：

```
<html>
  <body>
    <h1>User List</h1>
    <ul>
      {% for user in users: %}
      <li>
      {{user.firstname}} {{user.lastname}}
      </li>
      {% endfor %}
    </ul>
  </body>
</html>
```

如果要通过网页来修改数据，可使用WTForms(http://wtforms.readthedocs.io)为每个模型生成表单。WTForms是一个使用Python来定义表单并生成 HTML 表单的库，还负责从请求中提取数据，并在更新模型前校验数据。

Flask-WTF(https://flask-wtf.readthedocs.io/)项目特意为Flask封装了WTForms，并添

加了其他一些有用的集成功能，例如通过使用CSRF(Cross-Site Request Forgery)令牌，来增强表单的安全。



用户登录后，CSRF 令牌会确保恶意的第三方站点无法给应用发送有效的表单。第 7 章将详细解释 CSRF 的原理及其对应用安全的重要性。

下面的模块为 User 表实现了一个表单，你可从中了解 FlaskForm 的基本用法：

```
from flask_wtf import FlaskForm
import wtforms as f
from wtforms.validators import DataRequired

class UserForm(FlaskForm):
    email = f.StringField('email', validators=[DataRequired()])
    firstname = f.StringField('firstname')
    lastname = f.StringField('lastname')
    password = f.PasswordField('password')
    age = f.IntegerField('age')
    weight = f.FloatField('weight')
    max_hr = f.IntegerField('max_hr')
    rest_hr = f.IntegerField('rest_hr')
    vo2max = f.FloatField('vo2max')

    display = ['email', 'firstname', 'lastname', 'password',
               'age', 'weight', 'max_hr', 'rest_hr', 'vo2max']
```

代码中的 display 只是一个助手属性。它是一个包含表单字段的有序列表，模板在渲染表单时，会遍历其内容。此外，代码使用 WTForms 中基本的描述字段类为 User 数据库表创建表单。关于 WTForms 的字段的完整文档见 http://wtforms.readthedocs.io/en/latest/fields.html。

一旦创建完毕，即可在视图中使用 UserForm 达到两个目的：第一个是在 GET 请求中显示表单，第二个是当用户提交表单时，在 POST 请求中修改数据库中的相应内容。

```
@app.route('/create_user', methods=['GET', 'POST'])
def create_user():
    form = UserForm()
    if request.method == 'POST':
        if form.validate_on_submit():
            new_user = User()
```

```
            form.populate_obj(new_user)
            db.session.add(new_user)
            db.session.commit()
            return redirect('/users')
    return render_template('create_user.html', form=form)
```

UserForm 类包含一个用来校验 POST 请求中的数据的方法，以及一个把这些值序列化成 User 对象的方法。如果请求中的一些数据无效，表单的实例会将错误保存在 field.errors 列表中，以便接下来在模板中向用户显示这些错误。

create_user.html 模板遍历模板的字段列表，WTForm 负责将其渲染成合适的 HTML 标签：

```
<html>
  <body>
    <form action="" method="POST">
      {{ form.hidden_tag() }}
      <dl>
        {% for field in form.display %}
        <dt>{{ form[field].label }}</dt>
        <dd>{{ form[field]() }}</dd>
          {% if form[field].errors %}
          {% for e in form[field].errors %} <p>{{ e }}</p> {% endfor %}
          {% endif %}
        {% endfor %}
      </dl>
      <p>
      <input type=submit value="Publish">
    </form>
  </body>
</html>
```

form.hidden_tag()方法会渲染所有的隐藏字段，包括 CSRF 令牌。

一旦表单能正常工作，就可以很容易地在应用的所有表单中复用这个模式。

在 Runnerly 中，对于添加训练记录和竞赛的功能，我们需要复制这个模式来生成表单。模板中的表单部分是通用的，因此可在所有表单中复用，并可使用 Jinja 中的宏来替换。大部分工作将是对每个 SQLAlchemy 模型编写对应的 form 类。

有一个名为wtforms_alchemy(https://wtforms-alchemy.readthedocs.io/)的项目，使用它可基于SQLAlchemy模型自动生成表单代码。相对于与之前手动创建的UserForm，利用wtforms_alchemy来创建同样的类会简单得多，因为唯一的步骤是将其指向

SQLAlchemy模型：

```
from wtforms_alchemy import ModelForm

class UserForm(ModelForm):
    class Meta:
        model = User
```

但在实战中，通常会逐渐调整表单，一直到便于显式地编写为止。不过从 wtforms_alchemy 开始，来看看表单如何逐步演进也是一个解决方案。

总结一下迄今为止为了构建应用都做了什么：

- 使用 SQLAlchemy 创建了数据库模型(模型)。
- 创建了视图和表单，通过模型与数据库交互(视图和模板)。

在构建完整的单体方案前，还有两件事情：

- **后台任务**：用来定期检索 Strava 的跑步活动，以及生成月度报告。
- **身份验证和授权**：让用户登录，并限制其只能修改自己的信息。

4.2.3 后台任务

可通过轮询 Strava，定期(如每隔一小时)执行代码，从 Strava 获取所有跑步活动并添加到 Runnerly 数据库。月度报告也可在每个月生成汇总信息，然后通过电子邮件发送给用户。这两个功能都是 Flask 应用的一部分，用 SQLAlchemy 模型来完成。

但与其他用户请求不同，它们是后台任务，需要在 HTTP 请求和响应循环周期之外独立运行。

如果不使用简单的 cron 任务，一个在 Python Web 应用中运行重复性后台任务的流行方案是使用 Celery(http://docs.celeryproject.org)。它是一个在独立进程中执行某些工作的分布式任务队列。

为此，一个名为消息代理的中介负责在应用和Celery之间来回传递消息。例如，如果应用想让Celery运行某些任务，就会在消息代理中添加一个消息。Celery会轮询这个消息代理，并完成任务。

消息代理可以是任何存储消息并提供检索消息的服务。Celery项目能直接与 Redis(http://redis.io)、RabbitMQ(http://www.rabbitmq.com)、Amazon SQS(https://aws.amazon.com/sqs/)一起使用。它还为Python应用提供了一层抽象，以便应用能正常地发送消息和执行任务。

执行任务的部分被称为职程(worker)，Celery提供了Celery 类来启动它。在Flask应用中，可创建一个background.py模块来实例化一个Celery对象，然后使用@celery.task

装饰器来标记后台任务。

下例使用 stravalib(http://pythonhosted.org/stravalib)，为 Runnerly 中每个拥有 Strava 令牌的用户从 Strava 中抓取跑步活动：

```
    from celery import Celery
    from stravalib import Client
    from monolith.database import db, User, Run

    BACKEND = BROKER = 'redis://localhost:6379'
    celery = Celery(__name__, backend=BACKEND, broker=BROKER)
_APP = None

    def activity2run(user, activity):
        """"Used by fetch_runs to convert a strava run into a DB entry.
        """
        run = Run()
        run.runner = user
        run.strava_id = activity.id
        run.name = activity.name
        run.distance = activity.distance
        run.elapsed_time = activity.elapsed_time.total_seconds()
        run.average_speed = activity.average_speed
        run.average_heartrate = activity.average_heartrate
        run.total_elevation_gain = activity.total_elevation_gain
        run.start_date = activity.start_date
        return run

    @celery.task
    def fetch_all_runs():
        global _APP
        # lazy init
        if _APP is None:
            from monolith.app import app
            db.init_app(app)
            _APP = app
        else:
            app = _APP

        runs_fetched = {}
```

```
        with app.app_context():
            q = db.session.query(User)
            for user in q:
                if user.strava_token is None:
                    continue
                runs_fetched[user.id] = fetch_runs(user)


        return runs_fetched

    def fetch_runs(user):
        client = Client(access_token=user.strava_token)
        runs = 0
        for activity in client.get_activities(limit=10):
            if activity.type != 'Run':
                continue
            q = db.session.query(Run).filter(Run.strava_id == activity.id)
            run = q.first()
            if run is None:
                db.session.add(activity2run(activity)
                runs += 1

        db.session.commit()
        return runs
```

在该例中，后台任务会查找每个有Strava令牌的用户，将其最近10次跑步活动导入Runnerly。

这个模块是一个能从 Redis 代理接收任务并完成全部工作的 Celery 应用。假定在当前机器上已经运行了一个 Redis 实例，当使用 pip-install 命令安装了 Celery 和 Redis 的两个 Python 包后，就可以用 celery -A background worker 命令来运行这个模块。

这个命令会启动一个Celery职程服务器，将fetch_all_runs()函数注册为一个可调用的任务，并监听进入Redis的消息。

然后，在 Flask 应用中，可导入同样的 background.py 模块，然后直接调用被装饰的函数。此时得到一个类似 future 的对象，它将通过 Redis 来调用独立进程中的 Celery 职程来运行这个函数。

```
    from flask import Flask, jsonify

    app = Flask(__name__)
```

```
@app.route('/fetch')

def fetch_runs():
    from monolith.background import fetch_all_runs
    res = fetch_all_runs.delay()
    res.wait()
    return jsonify(res.result)
```

上例会等待任务完成，同时对/fetch 的调用也会等待任务完成后才会继续执行。当然，在 Runnerly 中，我们想用“发射后不管”(fire-and-forget)的方式来执行任务，而且不会调用.wait()方法，因为那样在每个用户上都会花费数秒的时间。

在某种意义上，由于 Celery 服务是由 Flask 应用通过 Redis 传递消息来调用的，因此可直接认为它是一个微服务。部署也很有趣，因为可将 Redis 服务器和 Celery 应用部署在其他服务器上。但由于执行后台任务的代码与应用的其他代码在同一个代码库中，因此这仍是一个单体设计。

运行后台职程要考虑的另一个问题是，有些任务需要被周期性地执行。此时可使用 Celery 的定期任务(Periodic Task，见 http://docs.celeryproject.org/en/latest/userguide/periodic-tasks.html)特性来充当调度程序，而不是让 Flask 应用每小时触发一次任务。

这种情况下，Flask 应用会采用与触发单个任务相同的方式，来调度这些定期任务。

1. Strava 令牌

在解决导入Strava内容的难题前，还有一个缺失的部分，那是为每个用户获取Strava令牌，并将其存储在数据库的用户表中。

这可通过OAuth2 dance来完成。在这个流程中，用户会被重定向到Strava为Runnerly授权；然后又被重定向到Runnerly，并带上OAuth2 code。此后可将这个code 转换成可保存的令牌。

Stravalib 库提供了一些助手程序来帮助完成这个 dance。首先是 authorization_url()方法，它会返回一个完整的 URL。将这个 URL 呈现给用户，即可开始 OAuth2 dance。

```
app.config['STRAVA_CLIENT_ID'] = 'runnerly-strava-id'
app.config['STRAVA_CLIENT_SECRET'] = 'runnerly-strava-secret'

def get_strava_auth_url():
    client = Client()
    client_id = app.config['STRAVA_CLIENT_ID']
    redirect = 'http://127.0.0.1:5000/strava_auth'
```

```
    url = client.authorization_url(client_id=client_id,
                                    redirect_uri=redirect)
    return url
```

这个例子中，redirect 变量是应用获取权限后 Strava 重定向的 URL。这个例子中，它是运行在本地应用中的地址。get_strava_auth_url()方法生成了返回给 Runnerly 用户的链接。

一旦用户在 Strava 网站中为 Runnerly 授权，/strava-auth 视图就会得到一个 code。然后用它来交换一个令牌。这个令牌会一直有效，可用来代表用户，给 Strava 发送请求。Strava 库的 Client 类有一个 exchange_code_for_token()方法来完成这个交换行为。

视图只将令牌保存到数据库的用户记录中。

```
@app.route('/strava_auth')
@login_required
def _strava_auth():
    code = request.args.get('code')
    client = Client()
    xc = client.exchange_code_for_token
    access_token = xc(client_id=app.config['STRAVA_CLIENT_ID'],
    client_secret=app.config['STRAVA_CLIENT_SECRET'], code=code)
    current_user.strava_token = access_token
    db.session.add(current_user)
    db.session.commit()
    return redirect('/')
```

在这个视图中，@login_required 和 current_user 是身份验证和授权的一部分，见下一节的介绍。

4.2.4 身份验证和授权

我们的单体应用已经接近完成了。

最后需要添加的功能是：给用户提供身份验证方式。Runnerly 需要知道当前要连接的用户，因为仪表盘会显示特定用户的数据。表单也需要安全保障。例如，不能让用户编辑其他用户的信息。

针对目前的单体方案，将实现一个简单的基本身份验证(https://en.wikipedia.org/wiki/Basic_access_authentication)模式。使用这个模式，用户通过 Authorization 请求头来发送凭证信息。从安全角度看，只要服务器使用 SSL，那么使用基本身份验证是没问题的。当通过 SSL 访问网站，整个请求会都会被加密(包括 URL 中都查询字符串)，

因此传输是安全的。

就密码而言，最简单的保护形式是确保不在数据库以明文形式存储密码，而将其保存在一个不可逆的哈希字符串中。如果服务器受到安全威胁，那么这个方式能将泄露密码的风险降至最低。对于身份验证过程，只需要在用户登录时，对传入的密码进行哈希处理，然后与保存在数据库中的字符串做比较。



传输层通常不是应用安全性的弱点。服务收到请求后会发生什么才重要。在身份验证过程中有一个时间窗口，攻击者会拦截密码(明文或哈希后的形式)。第 7 章将讨论减少这种攻击面的方法。

werkzeug提供了一些助手方法对密码做哈希，其中generate_password_hash()和check_password_hash()可集成到User 类中。

默认情况下，werkzeug 使用 PBKDF2(https://en.wikipedia.org/wiki/PBKDF2) 和 SHA-1 算法完成哈希，它使用盐值来哈希一个值，这是一种比较安全的方式。

下面通过添加设置和验证密码来扩展 User 类：

```
    from werkzeug.security import generate_password_hash,
check_password_hash
    class User(db.Model):
        __tablename__ = 'user'
        # ... all the Columns ...

        def __init__(self, *args, **kw):
            super(User, self).__init__(*args, **kw)
            self._authenticated = False

        def set_password(self, password):
            self.password = generate_password_hash(password)

        @property
        def is_authenticated(self):
            return self._authenticated

        def authenticate(self, password):
            checked = check_password_hash(self.password, password)
            self._authenticated = checked
            return self._authenticated
```

在数据库中创建新用户时，可使用 User 模型上的 set_password()方法来哈希和存储密码。验证密码时，可使用 authenticate()方法来比较哈希值。

有了这个机制后，Flask-Login(https://flask-login.readthedocs.io/)扩展提供了登录、退出和跟踪已连接的用户所需的一切，然后就可以修改应用的工作方式。

Flask-Login 提供两个函数，用于在当前的 session 中设置用户：login_user()和 logout_user()。调用login_user()方法时，会在session中保存用户ID，并在客户端设置一个cookie。应用会记住这个用户，并可在用户的下次请求中使用，直到用户退出。

为使用这个机制，需要在应用启动时创建一个 LoginManager 实例。

下面的代码登录和退出视图，并创建 LoginManager：

```
from flask_login import LoginManager, login_user, logout_user

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()
    if form.validate_on_submit():
        email, password = form.data['email'], form.data['password']
        q = db.session.query(User).filter(User.email == email)
        user = q.first()
        if user is not None and user.authenticate(password):
            login_user(user)
            return redirect('/')
    return render_template('login.html', form=form)

@app.route("/logout")
def logout():
    logout_user()
    return redirect('/')

login_manager = LoginManager()
login_manager.init_app(app)

@login_manager.user_loader
def load_user(user_id):
    user = User.query.get(user_id)
    if user is not None:
        user._authenticated = True
    return user
```

当Flask-Login需要将已经存储的用户ID转换成用户实例时，就会使用被@login_manager.user_loader装饰的方法。

身份验证工作会在login视图中通过调用user.authenticate()来完成，然后调用login_user(user)来修改session中的身份验证信息。

最后一件事情是保护一些视图以防未授权的访问。例如，未登录时，是不能访问用户编辑表单的。使用@login_required装饰的视图后，会拒绝任何来自未登录用户的访问，并返回 401 Unauthorized 错误。

@login_required 需要放在@app.route()调用之后：

```
@app.route('/create_user', methods=['GET', 'POST'])
@login_required
def create_user():
    # ... code
```

上面的代码中，@login_required会确保当前用户是有效的，并已通过身份验证。

然而，这个装饰器并不会处理访问的权限。权限处理不在Flask-Login项目范畴内，但可在Flask-Login上用其他扩展加以处理(如Flask-Principal，https://pythonhosted.org/Flask-Principal/)。

不过，以上方式对Runnerly的场景来说有点小题大作。Runnerly 用户中的一个特定角色是管理员。管理员拥有访问应用的超级权限，而普通用户只能访问自身的信息。

在 User 模型上添加 is_admin 的布尔标识后，即可创建与@login_required 类似的装饰器来检查这个标记。

```
def admin_required(func):
    @functools.wraps(func)
    def _admin_required(*args, **kw):
        admin = current_user.is_authenticated and current_user.is_admin
        if not admin:
            return app.login_manager.unauthorized()
        return func(*args, **kw)
    return _admin_required
```

同样，通过查看Flask-Login在应用上下文设置的current_user 变量，可执行更精细的权限验证。例如，可使用此方法允许用户更改自己的数据，但防止其修改其他用户的数据。

4.2.5 单体设计汇总

现在的单体设计还不错，也符合第一个开发迭代的目标。如第 3 章所述，应该使用 TDD 来构建一切。

这是构建在关系型数据库上的一个简单而整洁的实现，可与 PostgreSQL 或 MySQL 服务器一起部署。多亏了 SQLAlchemy 抽象，在日常的开发和测试中，可使用 SQLite3 作为数据库。

为构建这个应用，已经使用了下面的扩展和库：

- flask_sqlalchemy 和 SQLAlchemy：用来实现模型。
- Flask-WTF 和 WTForms：用来实现表单。
- Celery 和 Redis：用来实现后台处理和周期性任务。
- Flask-Login：用来管理身份验证和授权。

图 4-1 展示了整体设计。

一个典型的部署方案是，将 Flask 应用、Redis 服务器和 Celery 实例组合在一起，部署在同一个服务器上，并通过 Web 服务器(例如 Apache 或 nginx)来处理请求。数据库可部署在同一台机器，或者一个专属服务器上。

服务器可生成多个 Flask 进程和 Celery 进程，以处理更多请求和支持更多用户。

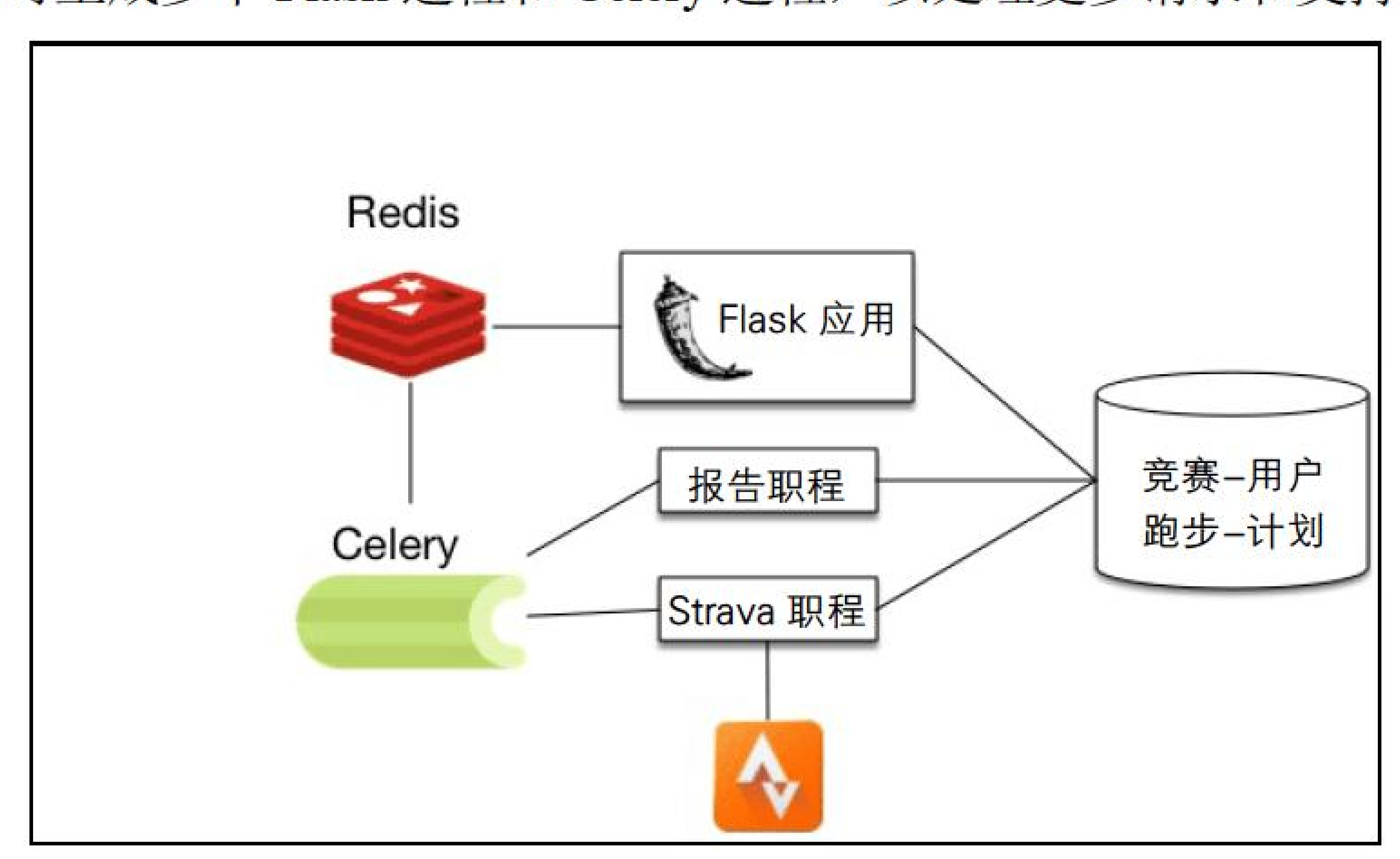


图 4-1　整体设计

当上面的部署方案满足不了服务器负载时，首先要考虑的是添加更多应用服务器，并为数据库和 Redis 代理添加专属服务器。

如有必要，可进入第三步，添加更多 Redis 和 PostgreSQL 实例。此时需要在选择最佳方式上多加思考，因为可能需要创建数据库的复制集或使用分片策略。



进入第三步时，使用一些开箱即用的方案可能更适合，如 Amazon SQS 和 Amazon RDS。第 11 章将讨论这个话题。

4.3　拆分单体

假定 Runnerly 还在使用先前的实现，但已拥有大量用户。期间添加了新功能，修复了漏洞，并且数据库也在稳定增长。

第一个要面对的问题是发送报告和访问 Strava 的后台进程。由于已有数千个用户，这些任务占用了大部分服务器资源，导致前端用户感觉网站变慢了。

显然，需要把这些后台任务转移到独立的服务器上来运行。对于使用 Celery 和 Redis 的单体应用来说，这不是问题，添加新的服务器用于后台任务即可。

但如果这样做，最令人担忧的是 Celery 职程需要导入 Flask 应用的代码来运行。所以在部署后台职程时，同时部署了整个 Flask 应用。这也意味着，每次修改应用，都需要更新 Celery 职程，以免修复过的错误再次出现。

这同样意味着，我们不得不在这个仅用来从 Strava 提取数据的服务器上，安装 Flask 应用所有的依赖。假如在模板中使用了 Bootstrap，那么也得在 Celery 职程服务器上部署它！

依赖会引起另一个问题："为什么 Celery 职程起初需要包含在 Flask 应用中？"在早期编写 Runnerly 时，这个设计非常优秀，但很显然，逐渐变得脆弱起来。

Celery 与应用的交互非常明确。Strava 职程需要：

- 获取 Strava 令牌
- 添加新的跑步活动

与使用 Flask 应用的代码的做法相反，Celery 职程可完全与之独立，并直接与数据库交互。

让 Celery 职程扮演一个独立微服务是拆分单体应用的非常重要的第一步——我们称之为 Strava 服务。负责构建报告的职程，可按独立运行的方式拆分，将其称为报告服务。这样，每个 Celery 进程能专注于单一类型的任务。

完成上面的拆分工作时，最大的设计问题是：这些微服务是直接访问数据库，还是通过一个充当服务与数据库的中介的 HTTP API 来调用？

直接访问数据库的方式看起来最简单，但也引入了其他问题。由于最初的 Flask 应用、Strava 服务和报告服务会共享一个数据库，因此数据库中的任何修改都会影响这三者。

如果有一个中介层，它给不同的服务公开了它们所需的信息，就会减少数据库依赖问题。如果设计得当，可在修改数据库模式时，保持 HTTP API 契约的兼容性。

Strava 和报告服务都是 Celery 职程，所以不需要对其设计任何 HTTP API。它们从 Redis 代理获取任务，然后使用封装了数据库调用的服务与其交互。我们称这个媒介为数据服务(Data Service)。

4.4 数据服务

图 4-2 描述了更新后的应用的组织形式。报告和 Strava 服务从 Redis 中获取任务，并与数据服务交互。

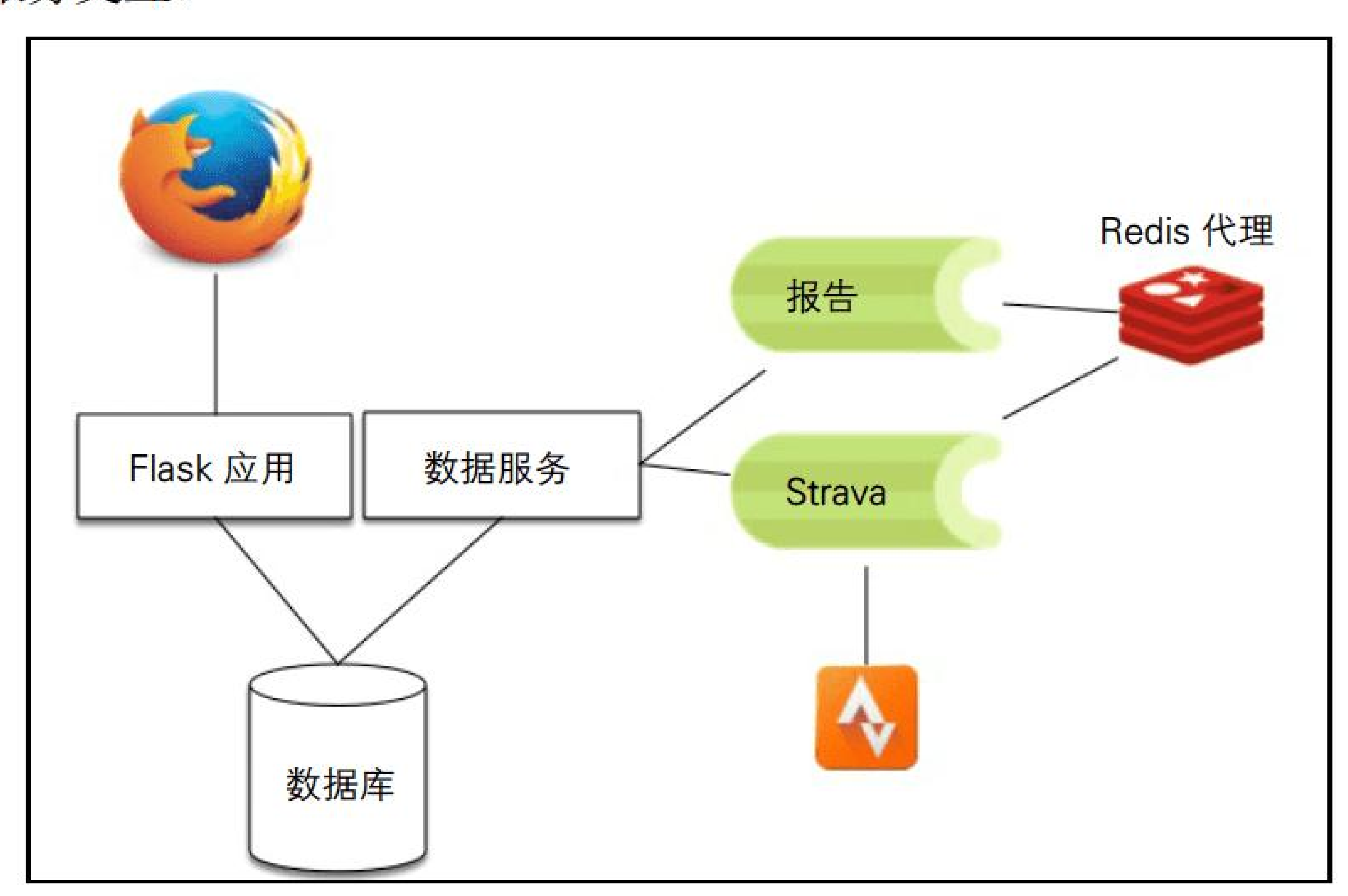


图 4-2　更新后的应用的组织形式

数据服务是一个封装了数据库操作的 HTTP API。数据库包含所有用户和跑步数据。仪表盘是前端应用，实现了 HTML 用户界面。

如果你不确定是否应当从主应用中拆分出一个新的微服务，就不要拆分它。

可通过 Redis 代理来传递 Celery 职程需要的一些信息，例如 Strava 服务所需的 Strava 令牌。

然后，对于报告服务，通过 Redis 来发送所有信息并不实际，因为所涉及的数据量非常大。如果一个跑步者每个月跑 30 次，那么更简单的做法是让报告服务直接从数据服务拉取数据。

数据服务的视图需要实现以下 API：

- 对于 Strava 服务——一个 POST API 用来添加跑步活动。

- 对于报告服务，需要：
  - 一个 GET API 来获取用户 ID 的列表。
  - 一个 GET API 来根据给定 ID 和月份获取跑步活动列表。

如你所见，HTTP API 非常小——我们期望尽量减少公开的入口数量。尽管跑步活动的结构会在多个服务之间共享，但我们还是需要尽可能减少公开的字段数量。

我们将采用 Open API 2.0 标准来实现服务。

4.5 使用 Open API 2.0

Open API 2.0 规范——也称为 Swagger(https://www.openapis.org/)——是采用 JSON 和 YAML 格式的简单描述语言，它列出所有 HTTP API 端点及其用法，以及传入和返回的数据结构。它假定服务器发送和接收 JSON 文档。

Swagger与XML Web服务时代里的WSDL(https://en.wikipedia.org/wiki/Web_Services_Description_Language)的目标相同，但轻量更轻，更直观。

下面是一个 Open API 描述文件的最简短示例，它定义了唯一的/api/users_ids 的入口，并支持使用 GET 方法来获取用户 ID 列表：

```
swagger: "2.0"
info:
  title: Runnerly Data Service
  description: returns info about Runnerly
  license:
    name: APLv2
    url: https://www.apache.org/licenses/LICENSE-2.0.html
  version: 0.1.0
basePath: /api
paths:
    /user_ids:
      get:
        operationId: getUserIds
        description: Returns a list of ids
        produces:
        - application/json
      responses:
        '200':
          description: List of Ids
```

```
schema:
  type: array
  items:
    type: integer
```

完整的 Open API 2.0 规范见 http://swagger.io/specification/。它非常详细，介绍了如何描述 API 的元信息、端点和使用的数据类型。

schema 部分描述的数据类型遵循 JSON 模式规范(http://json-schema.org/latest/json-schema-core.html)。上例定义了/get_ids端点返回一个整数列表。

可在规格说明中提供 API 的许多细节——例如请求中应该包含的请求头，某些响应中的 content-type 等。

使用 Swagger 描述 HTTP 入口点，能提供一些潜在好处：

- 许多 Open API 2.0 客户端能使用 API 的描述信息并完成一些有用的工作。例如为服务构建功能测试，或验证发给服务的数据。
- 它提供一个与语言无关的标准 API 文档。
- 服务器能根据规格说明来检查请求和响应。

一些 Web 框架甚至使用 Swagger 规格说明给微服务创建所有的路由和 I/O 数据检查。例如，Connexion(https://github.com/zalando/connexion)就为 Flask 提供了这些功能。

当人们使用 Swagger 构建 HTTP API 时，有两种思想流派：

- 规格说明先行(specification-first)：先创建 Swagger 规格说明文件，然后在此基础上使用规范中的信息来创建应用。这是 Connexion 背后的原则。
- 提取规格说明(specification-extracted)：使用代码来生成 Swagger 规格说明文件。例如，一些工具会通过读取视图的文档字符串(docstring)来生成它。

第一个方法的最大好处是：由于 Swagger 规格说明驱动了应用的开发，因此一定能及时更新。第二个方法也有价值，例如，在遗留项目中引入 Swagger 时就很有用。

如果通过第一个方法来实现 Flask 应用，Connexion 等框架会在较高层次上提供一些极好的助手程序。只需要传递规格说明文件、函数，Connexion 就能生成一个 Flask 应用。Connexion 使用 operationId 字段来解析函数与操作的对应关系。

使用这个方式时需要注意，Swagger 文件包含实现细节(指向 Python 函数的完整路径)，对于一个语言无关的规格说明来说，有一定的侵入性。还有一个自动解析器，会根据每个操作的路径和方法，来查找 Python 函数。这样，GET/api/users_ids 的实现就需要位于 api.users_ids.get()中。

flakon(见第 2 章)给出另一种方法。这个项目有个特殊的 Blueprint——SwaggerBlueprint，它不需要在规格中添加Python函数，也不会猜测操作对应的函数的位置。

这个自定义的 Blueprint 使用了 Swagger 规格文件，并提供一个与@api.route 类似的@api.operation 装饰器。这个装饰器使用 operationId 名称(而不是路由)作为参数，因此 Blueprint 可显式地将视图和正确路由连接起来。

下例创建一个 SwaggerBlueprint，并实现 getUserIds 操作：

```
from flakon import SwaggerBlueprint

api = SwaggerBlueprint('swagger', spec='api.yml')

@api.operation('getUserIds')
def get_user_ids():
    # .. do the work ..
```

重命名这段 Python 的实现代码或移动它的位置后，并不需要修改 Swagger 的规格说明。

除了上面提到的实现，数据服务API的剩余部分可参见Runnerly仓库(https://github.com/Runnerly)。

4.6　进一步拆分

到目前为止，我们已将后台任务的相关部分从单体应用中拆分出来，并为新的微服务添加了若干个 HTTP API 视图，以便与主应用交互。

由于新的 API 允许添加跑步活动，因此还有一个可拆分的部分——训练功能。

只要能生成新的跑步活动，这个功能就可独立运行。当用户想开始一个新的训练计划，主应用就可与训练服务交互，让其生成新的跑步活动。

作为另一个选择，为更好地隔离数据，这个方式也可保留下来：训练服务发布一个 API，这个 API 返回跑步活动的列表并使用特定的数据结构，正如 Strava API 返回的活动那样。主 Flask 应用能将它们转换成 Runnerly 的跑步活动。训练计划可在不依赖任何 Runnerly 用户的情况下工作——它被要求根据给定参数来生成一个训练计划。

但是，要以这个方式进行拆分，应当有合适的理由。例如：训练算法的代码是 CPU 密集型吗？它将来会成为一个完整的专家系统，被其他应用使用吗？训练功能将来需要其他数据来运行吗？



每次拆分新的微服务时，都有创建出臃肿应用的风险。

同样，竞赛功能也可能在某个时间点成为一个独立的微服务，因为竞赛列表可与Runnerly 数据完全独立。

图 4-3 展示了 Runnerly 的最终设计，包含四个微服务和主应用。在第 11 章，我们将看到如何抛弃主应用，使数据服务变成一个完整微服务，并构建一个 JavaScript 应用来集成一切。

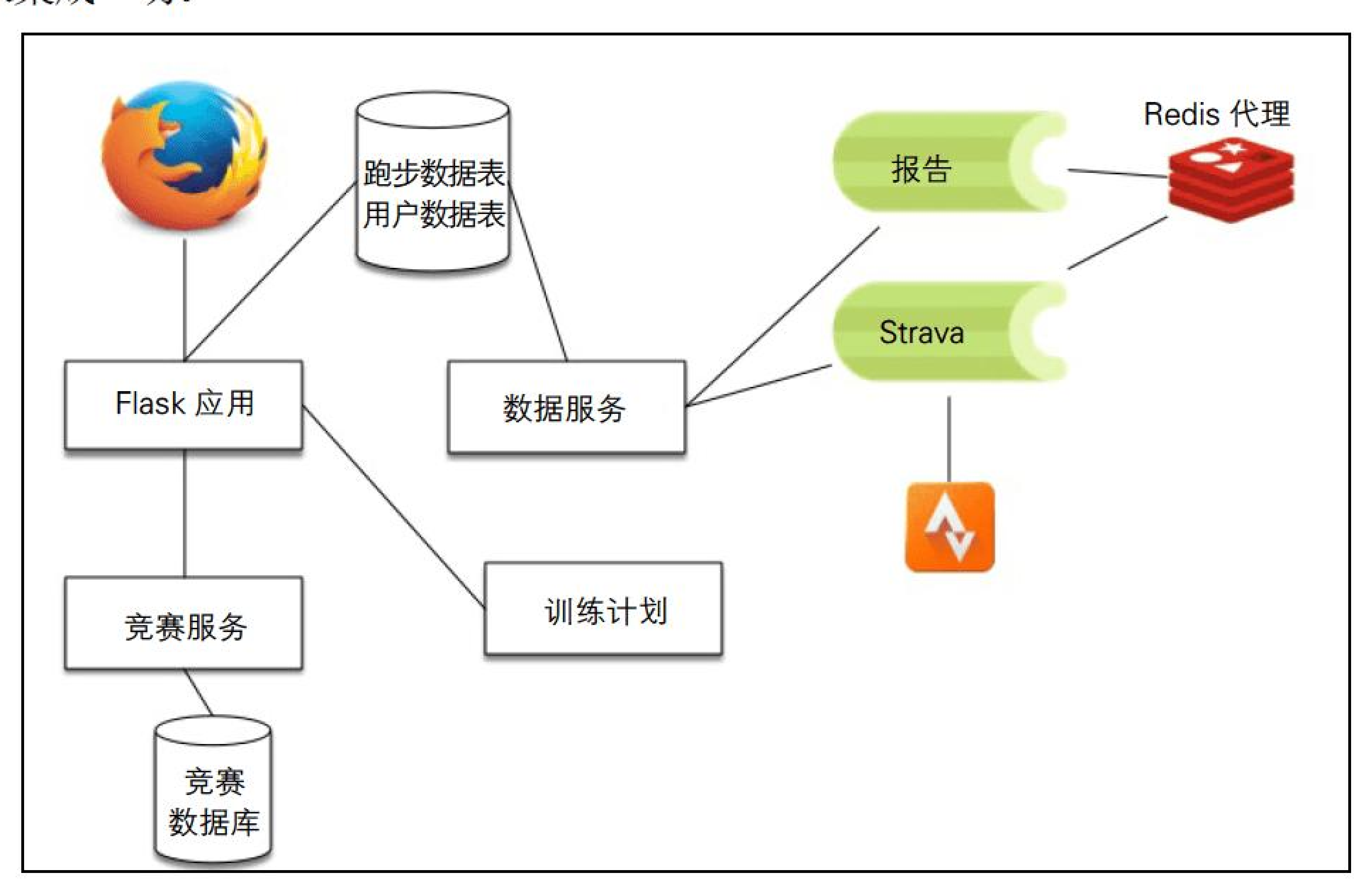


图 4-3　Runnerly 的最终设计

4.7　本章小结

Runnerly 应用是一个典型的 Web 应用，它与数据库和后端服务交互。最初几个迭代将其构建成一个单体应用。

本章演示了如何将单体逐渐拆分成微服务，以及 Celery 等工具在其中如何发挥作用。每一个能被拆分成独立 Celery 任务的后台进程都是潜在的微服务。

我们也讨论了 Swagger，它是一个帮助定义微服务之间的 API 的良好工具。

拆分进程应该是保守和渐进的，因为稍不留意，构建和维护微服务的代价就会超过拆分的好处。

如果你喜欢软件架构，这个应用的最后一个版本非常吸引人。它提供了用于部署和扩展 Runnerly 的许多选项。

然而，单一应用已变成需要交互的多个应用。图 4-3 的每个链路都可能是应用的弱点。例如，万一 Redis 宕机怎么办？或者，在处理过程中，如果数据服务和 Strava 服务之间的网络隔离，该怎么办？

加入架构中的每条网络链路都存在同样的问题。出问题时，需要能快速恢复。当修复一个宕机的服务时，我们需要知道问题出在哪里，以及如何解决问题。

以上问题都将在第 5 章中讨论。

第 5 章 与其他服务交互

在上一章中，将单体应用 Runnerly 拆分成多个微服务，因此增加了不同部分之间的网络交互。

当用户浏览主页面视图时，应用需要从数据库和竞赛服务中提取跑步和竞赛的列表。由于希望能立即显示结果，被请求触发的网络调用应当是同步的。

另一方面，Celery 职程正在后台完成职责，通过 Redis 代理异步地接受命令。

某些情况下，混合同步和异步调用也很有用。例如，当用户选择一个新的训练计划时，会显示有关该训练计划的一些信息，同时在后台触发并创建一系列新的跑步活动。

在 Runnerly 的未来版本中，可能有更多的服务间交互，一个服务的事件会触发其他服务的一系列反应。通过异步消息对系统的不同部分进行松耦合处理，这种方式可有效阻止服务间相互依赖。

任何情况下，最底线的要求是通过网络与其他服务进行同步或异步交互。这些交互应当是高效的，还要有应对计划来处理错误。

添加了更多网络连接时，产生的另一个问题是测试：当需要其他微服务时，如何在一个独立的微服务中进行测试？

本章将介绍：

- 一个服务如何以同步方式调用另一个服务，并让调用尽可能高效。
- 一个服务如何发起异步调用，如何通过事件与其他服务通信。
- 对于有网络依赖的服务，一些用来测试它们的技术。

5.1 同步调用

前面介绍过，微服务间的同步交互通过 RESTful HTTP API 来完成，API 使用 JSON 格式。

这是目前最常见的范式，因为 HTTP 和 JSON 是黄金标准。如果 Web 服务实现了接收 JSON 的 HTTP API，那么任意编程语言的开发者都能方便地使用它。

另一方面，遵循 RESTful 的结构并不是一个要求，而是一种有倾向的解释。关于使用 POST 与 PUT 响应的优缺点，互联网上有无数博客帖子在争论这个问题。

一些项目在 HTTP 上实现了 RPC(Remote Procedure Call，远程过程调用)API，这种方式不同于 REST API。在 RPC 方法中，重点是行为，行为是端点 URL 的一部分。在 REST 方法中，重点是资源，行为通过 HTTP 方法进行定义。

有些项目将两者混合使用，并未严格遵循一个标准。不过最重要的是，服务的行为应该是一致的，并有良好的文档。



本书主要依赖 REST 方式而非 RPC 方式，但也不是特别严格；另外，在 PUT 和 POST 的争论上也没有强烈倾向。

当微服务与其他服务进行交互时，发送和接受 JSON 是最简单方式，只需要微服务知道发送 HTTP 请求的入口点和需要传递的参数即可。

为此，只需要使用一个HTTP客户端。Python有一个内置的http.client模块，但requests库(https://docs.python-requests.org)有更好用的API，而且提供的内置功能会让你的工作更轻松。

requests 库中的 HTTP 请求是围绕 Session 概念构建的，使用它的最佳方式是创建一个 Session 对象，当每次与其他服务交互时会重用 Session 对象。

除其他事项外，Session 对象中还可保存身份验证信息以及一些默认的请求头信息，这些头信息是应用生成的所有请求都需要的。在下例中，Session 对象将自动创建正确的 Authorization 和 Content-Type 头信息：

```
from requests import Session

s = Session()
s.headers['Content-Type'] = 'application/json'
s.auth = 'tarek', 'password'

# doing some calls, auth and headers are all set!
s.get('http://localhost:5000/api').json()
```

```
s.get('http://localhost:5000/api2').json()
```

在一个与其他服务交互的 Flask 应用中，看一下如何让这个模式更通用。

5.1.1　在 Flask 应用中使用 Session

Flask 的 Application 对象有一个 extensions 映射关系，这个映射关系用来存储实用工具，例如连接器。在这个案例中，要用它存储 Session 对象。可创建一个函数，用来在 app.extensions 映射中初始化一个占位符，然后添加一个 Session 对象：

```
from requests import Session

def setup_connector(app, name='default', **options):
    if not hasattr(app, 'extensions'):
        app.extensions = {}

    if 'connectors' not in app.extensions:
        app.extensions['connectors'] = {}
    session = Session()

    if 'auth' in options:
        session.auth = options['auth']
    headers = options.get('headers', {})
    if 'Content-Type' not in headers:
        headers['Content-Type'] = 'application/json'
    session.headers.update(headers)

    app.extensions['connectors'][name] = session
    return session


def get_connector(app, name='default'):
    return app.extensions['connectors'][name]
```

在本例中，setup_connector()函数将创建一个 Session 对象，然后将它存放在应用的扩展映射中。创建的 Session 默认将 Content-Type 头信息设置成 application/json，这样就适合给基于 JSON 格式的微服务发送数据了。

一旦设置完成，即可通过 get_connector()函数在视图中使用 Session。在下例中，Flask 应用(运行在 5001 端口)将同步地调用微服务(运行在 5000 端口)来提供服务内容：

```
from flask import Flask, jsonify

app = Flask(__name__)
setup_connector(app)

@app.route('/api', methods=['GET', 'POST'])
def my_microservice():
    with get_connector(app) as conn:
        sub_result = conn.get('http://localhost:5000/api').json()
    return jsonify({'result': sub_result, 'Hello': 'World!'})

if __name__ == '__main__':
    app.run(port=5001)
```

对一个服务的调用将传播给其他服务：

```
$ curl http://127.0.0.1:5001/api
{
    "Hello": "World!",
    "result": {
        "Hello": "World!",
        "result": "OK"
    }
}
```

这种简单实现假设所有调用都很顺畅。但如果被调用的微服务发生延迟或等 30 秒后才返回，会发生什么？

默认情况下，响应就绪前，将无限期挂起请求；当调用微服务时，这并不是期望的行为。这种情况下，timeout 选项非常有用。如果在创建请求时使用它，当远程服务器未能及时回答时，它会抛出一个 ReadTimeout。

在下例中，如果请求挂起的时间超过 2 秒，将放弃调用：

```
from requests.exceptions import ReadTimeout

@app.route('/api', methods=['GET', 'POST'])
def my_microservice():
    with get_connector(app) as conn:
        try:
            result =
  conn.get('http://localhost:5000/api',timeout=2.0).json()
```

```
        except ReadTimeout:
            result = {}
    return jsonify({'result': result, 'Hello': 'World!'})
```

当然，当超时发生时，要做什么取决于服务逻辑。这个例子不加提示地忽略了这个问题，然后返回一个空结果。也许在其他场景下，需要抛出错误。无论哪种情况，如果要构建服务之间的强链接，就必须处理超时。

另一个可能发生错误的场景是：连接完全断开，或远程服务器根本无法访问。请求会进行多次尝试，最终抛出 ConnectionError，需要捕捉这个错误并进行处理：

```
from requests.exceptions import ReadTimeout, ConnectionError

@app.route('/api', methods=['GET', 'POST'])
def my_microservice():
    with get_connector(app) as conn:
        try:
            result = conn.get('http://localhost:5000/api',
                              timeout=2.).json()
        except (ReadTimeout, ConnectionError):
            result = {}
    return jsonify({'result': result, 'Hello': 'World!'})
```

始终使用超时参数是一个良好实践，一个更好的方式是在 Session 中将超时设置成默认参数，这样就不需要单独给每个请求设置超时参数了。

为此，requests 库提供了一种设置自定义传输适配器的方法，对于给定主机，可在传输适配器中定义 Session 将调用的行为。可用来创建一个通用的超时，但仍要提供 retries 参数，确定当服务器没有响应时，会完成多少次重试请求。

回到 setup_connector()函数。通过使用适配器，就可在所有请求中默认添加 timeout 和 retries 参数：

```
from requests.adapters import HTTPAdapter

class HTTPTimeoutAdapter(HTTPAdapter):
    def __init__(self, *args, **kw):
        self.timeout = kw.pop('timeout', 30.)
        super().__init__(*args, **kw)

    def send(self, request, **kw):
        timeout = kw.get('timeout')
```

```
        if timeout is None:
            kw['timeout'] = self.timeout
        return super().send(request, **kw)

def setup_connector(app, name='default', **options):
    if not hasattr(app, 'extensions'):
        app.extensions = {}

    if 'connectors' not in app.extensions:
        app.extensions['connectors'] = {}
    session = Session()

    if 'auth' in options:
        session.auth = options['auth']

    headers = options.get('headers', {})
    if 'Content-Type' not in headers:
        headers['Content-Type'] = 'application/json'
    session.headers.update(headers)

    retries = options.get('retries', 3)
    timeout = options.get('timeout', 30)
    adapter = HTTPTimeoutAdapter(max_retries=retries, timeout=timeout)
    session.mount('http://', adapter)
    app.extensions['connectors'][name] = session

    return session
```

每次向 HTTP 服务发出一个请求时，session.mount(host, adapter)函数将告诉请求使用 HTTPTimeoutAdapter。这种情况下，在 session 上挂载的键为"http://"的主机会成为一个包罗万象的容器。

对于 mount()函数，美妙之处在于 session 的行为可根据每个服务上的应用逻辑进行调整。例如，当需要设置一些自定义的超时和重试次数时，可在适配器上为特定主机加载另一个实例：

```
adapter2 = HTTPTimeoutAdapter(max_retries=1, timeout=1.)
session.mount('http://myspecial.service', adapter2)
```

幸亏有了这个模式，单请求的 Session 对象能在应用中实例化，然后通过它与其

他 HTTP 服务进行交互。

5.1.2　连接池

Requests 库底层使用 urllib3 库，将为每个目标主机创建一个连接池，当代码请求主机时，会复用连接池中的连接器。

换句话说，如果你的服务调用其他多个服务，不必担心这些服务连接的回收工作；requests 库负责处理回收。

Flask 是一个同步框架，如果用单一线程运行时(这是默认行为)，requests 库的连接池并没有太多帮助，因为各个调用一个接一个地执行。Requests 库只针对每个远程主机保持一个连接。

但若用多线程运行 Flask 应用，并有很多并发连接，连接池可有效地控制对其他服务的连接数量。你肯定不期望应用打开无限的同步连接去访问某个服务，这极易导致灾难。

HTTPTimeoutAdapter 类可用来控制连接池的增长。这个类继承自 HTTPAdapter，而 HTTPAdapter 封装了 urllib3 连接池的参数。

在构造函数中，可传递下列参数：

- pool_connection：确定可同时打开多少个连接。
- pool_maxsize：确定连接池的最大连接数量。
- max_retries：确定每个连接的最大重试次数。
- pool_block：当pool_maxsize到达上限时，确定是否阻塞连接。如果设置为False，即便连接池满了，也会创建一个新连接，但不会添加到连接池中。如果设置为True，连接池满后将不会创建新连接。对于最大化一个主机的连接数量，这个参数非常有用。

例如，如果应用在一个允许多线程的 Web 服务器上，适配器能持有 25 个并发连接：

```
adapter = HTTPTimeoutAdapter(max_retries=retries,
                             timeout=timeout, pool_connections=25)
```

提高服务性能的好方法是支持多线程，但多线程也伴随很多显著风险。通过自身的本地线程机制，Flask确保每个线程都能获取flask.g(全局的)、flask.request或flask.response版本，所以不需要处理线程安全问题；但视图被多个线程并发访问时，需要注意视图上会发生什么。

如果不共享 flask.g 以外的任何状态，而只是调用 Request 会话，这种方法也是可

行的。请求的会话不是线程安全的，所以每个线程需要单独一个会话。

对共享的状态做任何改变时，若未使用合适的锁机制来避免锁竞争问题，那就麻烦了。当视图太复杂而无法确保线程安全时，最安全的方式是运行单线程并引发多个进程。这时，每个进程都会执行一个 Request 会话；这个会话存在到外部服务的单一连接，并对调用进行序列化处理。

序列化是同步框架的限制因素，它迫使我们采用引发更多进程的方式进行部署，这种方式占用的内存更多，或需要使用诸如 Gevent 的隐式异步工具进行序列化。

不论哪种情况，只要单线程应用能迅速响应，将能大幅缓解这个限制。

有一个方法能提高应用调用其他服务的速度：确保使用 HTTP 缓存头。

5.1.3 HTTP 缓存头

在 HTTP 协议中，有几种缓存机制都可告知客户端：它试图访问的页面从上次访问以来没有任何更改。在我们的微服务中，可在所有只读 API 端点(如 GET 和 HEAD)上进行缓存。

最简单的实现方法是在响应中返回一个 ETag 头。ETag 的值是一个字符串，可将其视为客户端试图获取资源的版本信息。它可以是一个时间戳、一个增量的版本号或一个哈希串。由服务器决定其中的内容。但原则上它在响应值中应该是唯一的。

与 Web 浏览器一样，当客户端重取包含此类头信息的响应时，客户端会构建一个本地字典进行缓存，响应内容和 ETags 值作为 value，URL 作为 key。

当发起一个新请求时，客户端可查找字典，然后在请求的头信息 If-Modified-Since 中传递一个 ETag 值。如果服务器返回 304 响应，意味着响应没有发生改变，客户端可使用之前存储在本地字典的信息。

这种机制极大地缩短了服务器的响应时间。当内容没有改变时，可立刻返回一个空的 304 响应。由于 304 响应没有内容，所以网络传输的数据量也很小。

有一个配合 Request 会话使用的项目叫做 CacheControl(http://cachecontrol.readthedocs.io)，它透明地实现了缓存功能。

对于前面的例子，只需要将 HTTPTimeoutAdapter 的父类从 request. adapters. HTTPAdapter 替换成 cachecontrol.CacheControlAdapter 就可以激活缓存了。

当然，这意味着调用的服务应该通过添加合适的 ETag 来实现缓存行为。

由于缓存逻辑取决于数据的性质，而数据是由服务管理的，因此缓存问题并没有通用的解决方案。

规则是给每个资源标记版本信息，并在数据改变时修改它。下例中，Flask 应用使用当前服务器时间来创建 ETag 值，并与用户实体进行关联。ETag 值是自纪元以来经

历的时间(以毫秒为单位)，存储在修改后的字段中。

get_user()方法从_USERS 字典返回用户实体，然后通过 resp.set_etag 设置 ETag 值。当视图收到调用时，会查找 If-None-Match 头信息，然后与一个标记用户是否修改的字段进行比较，如果相同，则返回 304 响应：

```
import time
from flask import Flask, jsonify, request, Response, abort

app = Flask(__name__)

def _time2etag(stamp=None):
    if stamp is None:
        stamp = time.time()
    return str(int(stamp * 1000))

_USERS = {'1': {'name': 'Tarek', 'modified': _time2etag()}}

@app.route('/api/user/<user_id>', methods=['POST'])
def change_user(user_id):
    user = request.json
    # setting a new timestamp
    user['modified'] = _time2etag()
    _USERS[user_id] = user
    resp = jsonify(user)
    resp.set_etag(user['modified'])
    return resp

@app.route('/api/user/<user_id>')
def get_user(user_id):
    if user_id not in _USERS:
        return abort(404)
    user = _USERS[user_id]

    # returning 304 if If-None-Match matches
    if user['modified'] in request.if_none_match:
        return Response(status=304)

    resp = jsonify(user)
```

```
    # setting the ETag
    resp.set_etag(user['modified'])
    return resp
if __name__ == '__main__':
    app.run()
```

当客户端针对一个用户执行 POST 操作时，change_user()视图会设置一个新的修改标识。在接下来的客户端会话中，将执行用户更改，并确保提供新的 ETag 值时会得到 304 响应：

```
$ curl http://127.0.0.1:5000/api/user/1
{
    "modified": "1486894514360",
    "name": "Tarek"
}

$ curl -H "Content-Type: application/json" -X POST -d
'{"name":"Tarek","age":40}' http://127.0.0.1:5000/api/user/1
{
    "age": 40,
    "modified": "1486894532705",
    "name": "Tarek"
}

$ curl http://127.0.0.1:5000/api/user/1
{
    "age": 40,
    "modified": "1486894532705",
    "name": "Tarek"
}

$ curl -v -H 'If-None-Match: 1486894532705'
http://127.0.0.1:5000/api/user/1
< HTTP/1.0 304 NOT MODIFIED
```

这里只是展示一个示例实现，可能无法正常运行，因为这里依靠服务器时钟来存储 ETag 值，当有多个服务器时，要确保时钟不会被改回去；可使用诸如 ntpdate 的服务来保证服务器时钟总是同步的。

如果两个请求在同一毫秒内更改同一实体，则会导致锁竞争问题。基于不同的应用，这要么不是一个问题，要么就是大问题。一个干净的方案是让数据库系统直接处理修改后的字段，然后确保修改是通过序列化的事务完成的。

一些开发者使用哈希函数作为 ETag 值，这是因为在分布式架构中很容易计算，且不会产生任何时间戳问题。但计算哈希是有时间成本的，需要将整个实体的信息推送给服务器计算，这可能导致返回真实数据很慢。但若在数据库中有专门的表来管理哈希，这将是一个快速返回 304 响应的解决方案。

前面提到，并没有实现高效 HTTP 缓存逻辑的通用解决方案，但如果客户端会大量发起读请求，是值得使用缓存的。

当必须返回一些数据时，有多种提高效率的方法，下一节将进行介绍。

5.1.4　改进数据传输

虽然 JSON 相当冗长，但当需要与数据进行交互时，冗长是有意义的。所有内容都是清晰的明文且易于阅读，就像普通 Python 字典和列表一样。

但从长远看，使用 JSON 格式发送 HTTP 请求和响应会增加一些带宽开销。从 Python 对象到 JSON 结构数据的序列化和反序列化也会增加 CPU 开销。

通过压缩或转换成二进制格式，可减少传输数据的大小，并加快处理速度。

1. GZIP 压缩

要减少占用的带宽，最简单的方法是使用 GZIP 压缩，这样传输的数据总量会变小。诸如 Apache 或 nginx 的 Web 服务器原生地支持在传输时压缩响应，这很好地避免了在 Python 级别执行临时压缩。

例如，对于通过 Flask 应用在 5000 端口上产生的响应，下面代码示例中的 nginx 配置可对类型是 application/json 的任何响应进行压缩：

```
http {
    gzip on;
    gzip_types application/json;
    gzip_proxied any;
    gzip_vary on;

    server {
    listen       80;
    server_name  localhost;
    location / {
```

```
            proxy_pass http://localhost:5000;
        }
    }
```

从客户端，制作一个包含 Accept-Encoding: gzip 头信息的 HTTP 请求，把请求发送给位于 localhost:8080 的 nginx 服务器，服务器代理位于 localhost:5000 的应用，将触发压缩请求。

```
$ curl http://localhost:8080/api -H "Accept-Encoding: gzip"
<some binary output>
```

在 Python 中，request 库的 response 自动解压 gzip 编码格式的响应，所以当你的服务调用其他服务时，不需要担心这个问题。解压数据会增加一些 CPU 计算，但 Python 中的 gzip 模块依赖的是 zlib，zlib 的解压缩速度很快(也非常出色)。

```
>>> import requests
>>> requests.get('http://localhost:8080/api', headers=
{'Accept-Encoding':
'gzip'}).json()
{'Hello': 'World!', u'result': 'OK'}
```

要压缩发送给服务器的数据，可使用 gzip 模块并指定一个 Content-Encoding 头：

```
>>> import gzip, json, requests
>>> data = {'Hello': 'World!', 'result': 'OK'}
>>> data = bytes(json.dumps(data), 'utf8')
>>> data = gzip.compress(data)
>>> headers = {'Content-Encoding': 'gzip'}
>>> requests.post('http://localhost:8080/api',
...                 headers=headers,
...                 data=data)
<Response [200]>
```

然而，这种情况下，将在 Flask 应用中获得压缩的内容，除非在 nginx 中使用 Lua 实现了解压缩处理，否则需要在 Python 代码中解压。另一种方法是使用 mode_deflate 模块及其 SetInputFilter 配置项在 Apache 服务器上进行解压。

小结一下，使用 Apache 和 nginx 可很容易地设置 GZIP 压缩；通过设置正确的头信息，Python 客户端也能从中获益。如果不使用 Apache 处理 GZIP 压缩，则有点麻烦，要在 Python 代码中或其他地方实现对传入数据的解压。

要进一步减小 HTTP 请求/响应负载包的大小，相对于用 GZIP 压缩的 JSON 负载

包，另一个选项是转换成二进制负载包。这样做的好处是不需要解压数据，并能提高速度。但总体来看，这种压缩的效果并不理想。

2. 二进制格式

如果微服务需要处理大量数据，使用替代格式是一个有吸引力的方案，这种方式不需要依赖 GZIP 即可提高性能并减少网络带宽，不过使用替代方案并非总与提高性能相关。

有两种广泛使用的二进制格式工具：Protocol Buffers(protobuf)和 MessagePack。

Protocol Buffers(https://developers.google.com/protocol-buffers)要求解释转换成某种模式的数据，该模式将用来为二进制内容编写索引。

由于所有要转换的数据都按一种模式来描述，并需要学习一个新的领域特定语言(Domain Specific Language)，因此会增加一些工作量。

下例来自 protobuf 文档：

```
package tutorial;

message Person {
   required string name = 1;
   required int32 id = 2;
   optional string email = 3;

   enum PhoneType {
      MOBILE = 0;
      HOME = 1;
      WORK = 2;
   }

   message PhoneNumber {
      required string number = 1;
      optional PhoneType type = 2 [default = HOME];
   }

   repeated PhoneNumber phones = 4;
}

message AddressBook {
   repeated Person people = 1;
}
```

这并不像 Python 模式的数据，更像是数据库模式。虽然描述被转移的数据是一种良好实践，但在微服务中，如果已经有了 Swagger 定义，这种模式可能有点多余。

MessagePack(http://msgpack.org/)是另一种无模式的二进制工具，只需要通过调用一个函数就能轻松实现压缩和解压。

这是一个简单的 JSON 替代方式，大部分语言中都有实现。msgpack Python 库(通过 pip install msgpack-python 命令安装)提供了与 JSON 类似的集成。

```
>>> import msgpack
>>> data = {"this": "is", "some": "data", 1: 2}
>>> msgpack.dumps(data)
b'x83x01x02xa4thisxa2isxa4somexa4data'
>>> msgpack.loads(msgpack.dumps(data))
{1: 2, b'this': b'is', b'some': b'data'}
```

注意，数据序列化后，字符串被转换为二进制，然后要用默认序列器进行反序列化。当需要保持原始类型时，这是需要留意的问题。

很明显，MessagePack 比 protobuf 更简单。但哪一个压缩更快，并能提供最佳的压缩比率则取决于数据。少数情况下，普通 JSON 可能比二进制能更快速地序列化。

在压缩方面，预计 MessagePack 能达到 10%~20%的压缩比例，但若 JSON 里包含很多字符串(这种情况在微服务中很常见)，GZIP 是更好的选择。

在下例中，一个 87KB 的庞大 JSON 中包含很多字符串，它将用 MessagePack 方式进行转换，然后用 GZIP 对转换前和转换后的数据进行压缩：

```
>>> import json, msgpack
>>> with open('data.json') as f:
...     data = f.read()
...
>>> python_data = json.loads(data)
>>> len(json.dumps(python_data))
88983
>>> len(msgpack.dumps(python_data))
60874
>>> len(gzip.compress(bytes(json.dumps(data), 'utf8')))
5925
>>> len(gzip.compress(msgpack.dumps(data)))
5892
```

使用MessagePack减少了负载量，但通过粉碎方式，GZIP能做到比JSON和MessagePack格式小 15 倍。

很明显，不论使用哪种格式，如果 Web 服务器不处理解压缩问题，使用 GZIP 将是减少负载量大小的最佳方式。在 Python 中使用 gzip.uncompress()来解压缩非常容易。

至此，对于使用 MessagePack 还是 JSON 问题上，二进制格式通常更快，对 Python 也更友好。例如，如果传递的 Python 字典带有整数 key，JSON 会将整数转换成字符串，但 MessagePack 会转换成正确格式：

```
>>> import msgpack, json
>>> json.loads(json.dumps({1: 2}))
{'1': 2}
>>> msgpack.loads(msgpack.dumps({1: 2}))
{1: 2}
```

在日期格式上，两个方法都存在问题：在 JSON 和 MessagePack 中，DateTime 对象并不能直接序列化，需要通过代码来转换。

无论如何，在微服务世界里，JSON 是最广泛接受的标准，而坚持使用大众标准的一个小烦恼是：只能使用字符串 key，需要对日期类型进行额外处理。



在 Python 中，除非所有服务都具有优良的结构，且能让序列化步骤尽快执行，否则坚持使用 JSON 可能更简单。

5.1.5　同步总结

快速回顾一下本节中关于同步调用的内容：

- Requests 库可作为 HTTP 客户端调用其他服务。它提供了处理超时和错误的功能，还有一个连接池。
- 调用其他服务时，使用多线程可提高微服务的性能，这是因为 Flask 是一个同步框架。但使用多线程存在危险。可使用诸如 Gevent 的解决方案。
- 实现 HTTP 缓存头是提高重复数据请求速度的好方法。
- GZIP 压缩是一种减少请求和响应数据大小的有效方法，且易于设置。
- 二进制协议是一个有吸引力的 JSON 替代方案，但可能并不好用。

下一节将介绍异步调用。

5.2 异步调用

在微服务架构中，当原来单个应用中执行的进程现在变成多个微服务时，异步调用是很重要的基础角色。

使用异步调用可像微服务内部的单线程或单进程那样简单，会在不干涉 HTTP 请求-响应过程的情况下，完成一些工作。

但从同一个 Python 进程直接做任何事情都不十分可靠。如果进程崩溃或重启将发生什么？如果这样构建微服务又如何扩展后台任务呢？

更可靠的方法是发送一个消息然后由另一个程序接收并处理，这样可让微服务聚焦在它的目标上：给客户端提供响应服务。

第 4 章介绍了如何使用 Celery 构建一个微服务，它能与诸如 Redis 或 RabbitMQ 的消息代理进行协作。在那个设计中，在一个新消息加入 Redis 队列前，Celery 职程会阻塞队列。

还有一种方法可在服务间交换消息，而且职程不会阻塞队列。

5.2.1 任务队列

Celery 职程使用的这个模式是 push-pull 队列，服务将消息推送到特定队列，其他端点的职程会捡起它们然后执行一个动作。每个任务都是单一职程。如图 5-1 所示。

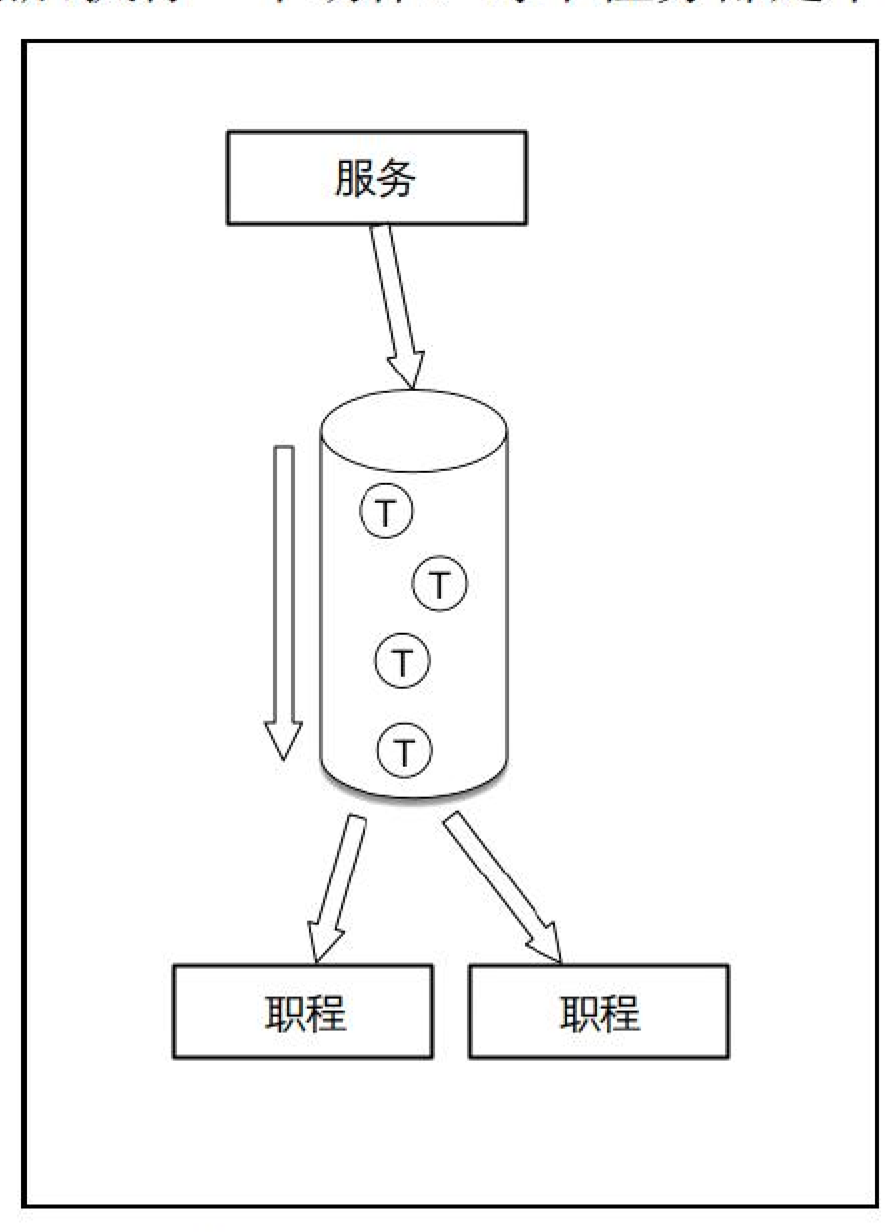


图 5-1 push-pull 队列

这里没有双向通信，发送者在队列中保存一个消息就离开了。下一个可用的职程将获得下一条消息。

若想执行一些异步并行任务，这种盲目的单向消息传递方式是完美的，易于扩展。

此外，一旦发送者已确认消息被添加到代理中，就可让诸如 RabbitMQ 的消息代理对消息进行持久化。换言之，即便所有职程离线，也不会丢失队列中的任何消息。

5.2.2　主题队列

任务队列模式的变体是主题模式。在这种模式下，与职程盲目地接收每个队列新增消息不同的是，职程只需要订阅特定主题的消息。主题就是消息的标签，职程可从队列中捡起消息，然后判断是否匹配主题来进行过滤。

在我们的微服务中，这意味着可以有特定的职程，它们都注册到消息代理上，然后获取添加到队列中的消息子集。

Celery 是构建任务队列的卓越工具，不过对于复杂消息，我们需要使用另一个工具，见图 5-2。

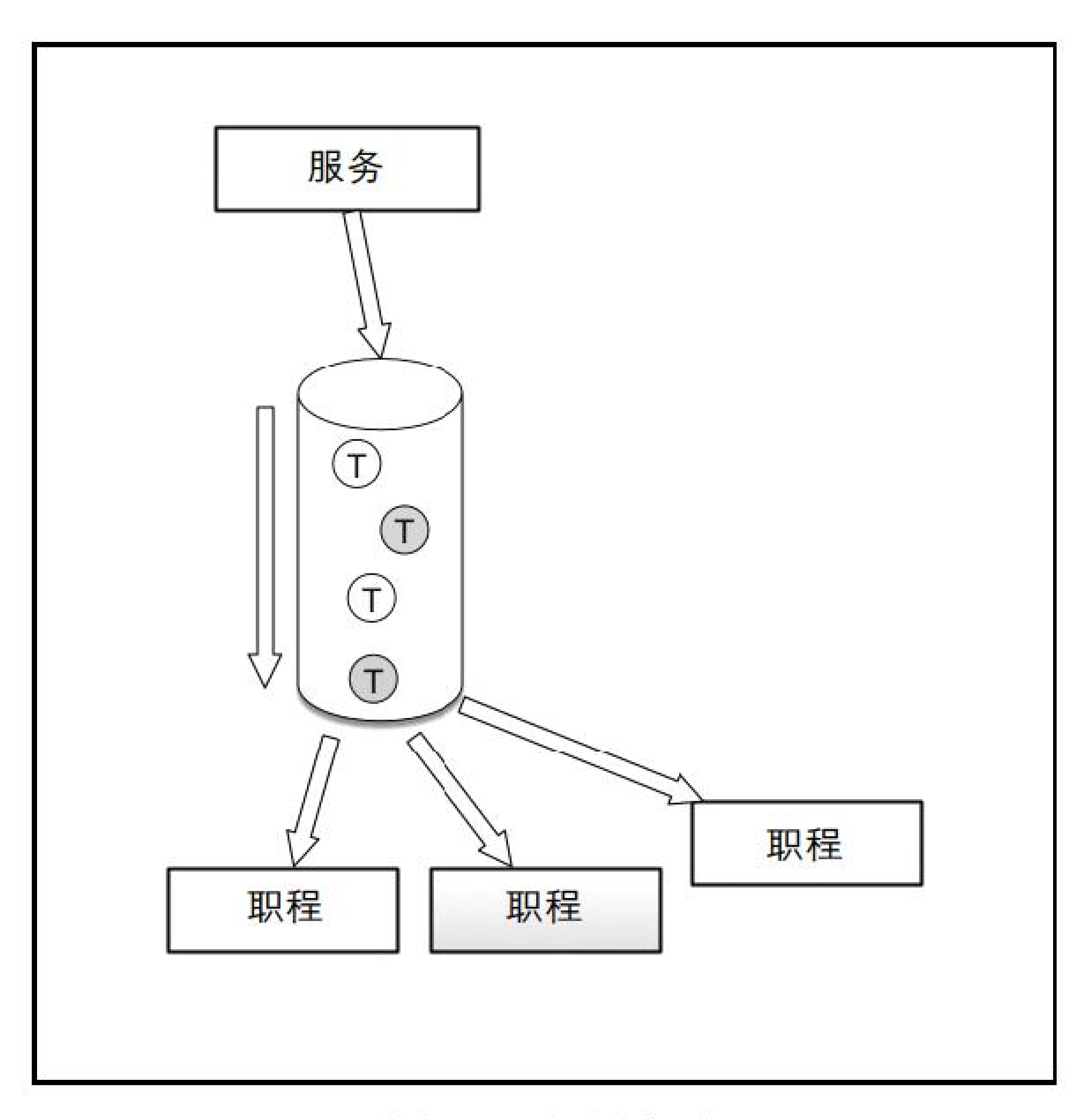


图 5-2　主题队列

要实现复杂消息模式，好消息是可使用 RabbitMQ 消息代理，它能与 Celery 以及其他库一起协同工作。

要安装 RabbitMQ 代理，可从下载页面 http://www.rabbitmq.com/download.html 开始。RabbitMQ 代理是一个 TCP 服务器，在内部管理队列，并通过 RPC 调用将消息从发布者分发到订阅者服务器。和 Celery 一起使用只是这个系统所提供的一小部分功能。

RabbitMQ实现了高级消息队列协议(Advanced Message Queuing Protocol，AMQP)。这个协议的相关详细说明可参见http://www.amqp.org/，它是由行业内多个大公司开发了多年的完整标准。

AMQP 包含以下概念：

- **队列**是持有消息的接收者，并等待消息消费者选择它们。
- **交换器**是入口，供发布者将新消息添加到系统。
- **绑定器**定义如何将消息从交换器路由到队列。

对于主题队列，需要设置一个交换器，这样 RabbitMQ 会接受新消息，并设置所有队列，职程会从队列中捡出消息。在中间过程，使用绑定器把消息路由到不同主题的队列上。

假设设置了两个职程，一个接受竞赛的消息，另一个接受训练的消息。

每次竞赛的消息都会打上标签 race.id，这里 race 是固定前缀，id 是竞赛的唯一标识。类似地，对训练消息也会打上标签 training.id。

通过安装 RabbitMQ，可使用 rabbitmqadmin 命令行创建所有必要的队列组件：

```
    $ rabbitmqadmin declare exchange name=incoming type=topic
    exchange declared

    $ rabbitmqadmin declare queue name=race
    queue declared

    $ rabbitmqadmin declare queue name=training
    queue declared

    $ rabbitmqadmin declare binding source="incoming"
destination_type="queue"
    destination="race" routing_key="race.*"
    binding declared

    $ rabbitmqadmin declare binding source="incoming"
destination_type="queue"
    destination="training" routing_key="training.*"
    binding declared
```

在这个设置中，每个消息都被发送给 RabbitMQ，如果主题以 race.开头，消息会被推送到 race 队列；如果以 training.开头，则被推送到 training 队列。

可使用 Pika(https://pika.readthedocs.io)在代码中与 RabbitMQ 进行交互，这是一个 Python RPC 客户端，实现了针对 Rabbit 功能的所有 RPC 接口。



通过 Pika 完成的所有事情都可在命令行上使用 rabbitmqadmin 实现。可获取所有队列的状态，可发送和接收消息，以及检查队列中的内容。使用 rabbitmqadmin 是实验消息设置的极佳方法。

下面的脚本演示如何向 RabbitMQ 的传入交换器发布两条消息。一条是 race.34，另一条是 training.12：

```
from pika import BlockingConnection, BasicProperties

# assuming there’s a working local RabbitMQ server with a working
  guest/guest account
def message(topic, message):
    connection = BlockingConnection()
    try:
        channel = connection.channel()
        props = BasicProperties(content_type='text/plain',
                                delivery_mode=1)
        channel.basic_publish('incoming', topic, message, props)
    finally:
        connection.close()
# sending a message about race 34
message('race.34', 'We have some results!')

# training 12
message('training.12', "It's time to do your long run")
```

这些 RPC 调用最终将在竞赛和训练队列中分别添加一条消息。等待竞赛消息的职程脚本如下所示：

```
import pika

def on_message(channel, method_frame, header_frame, body):
    race_id = method_frame.routing_key.split('.')[-1]
    print('Race #%s: %s' % (race_id, body))
    channel.basic_ack(delivery_tag=method_frame.delivery_tag)

print("Race NEWS!")
```

```
connection = pika.BlockingConnection()
channel = connection.channel()
channel.basic_consume(on_message, queue='race')
try:
    channel.start_consuming()
except KeyboardInterrupt:
    channel.stop_consuming()

connection.close()
```

注意，Pika 会向 RabbitMQ 返回一个关于消息的 ACK，这样一旦职程成功完成，就能安全地从队列中移除消息。

以上代码的输出结果为：

```
$ bin/python pika_worker.py
Race NEWS!
Race #34: b'We have some results!'
```

AMQP提供了多种交换信息的模式。教程页面http://www.rabbitmq.com/getstarted.html上列举多个示例，都是使用Python和Pika实现的。

要在微服务中集成这些示例，消息发布者部分的实现很简单。Flask 应用可使用 pika.BlockingConnection 来创建一个到 RabbitMQ 的同步连接，然后通过它发送消息。有一个 pika-pool(https://github.com/bninja/pika-pool)项目实现了简单的连接池，通过它可轻松管理 RabbitMQ 连接，当通过 RPC 发送消息时，不需要每次都建立或断开连接。

另一方面，对于消息消费者，在微服务中进行集成就有没那么容易了。

Pika 可被嵌入事件轮询器中，这个轮询器和 Flask 应用是同一个进程，当收到消息时会触发一个函数。这个功能对一个异步框架没啥问题，但是对 Flask 应用，则需要在另一个线程或进程中使用 Pika 客户端执行代码。这样做的原因是，当 Flask 收到一个 request 时，事件轮询器可能会被阻塞。

要使用 Pika 客户端和 RabbitMQ 进行交互，最可靠的方法是有一个独立的 Python 应用，以 Flask 微服务的身份消费消息，然后执行同步 HTTP 调用。这种方式增加了另一个中介，但有能力确保消息被成功接收。通过在本章前面所学的关于请求的所有技巧，我们能构建可靠的桥接：

```
import pika
import requests
from requests.exceptions import ReadTimeout, ConnectionError
FLASK_ENDPOINT = 'http://localhost:5000/event'
```

```
def on_message(channel, method_frame, header_frame, body):
    message = {'delivery_tag': method_frame.delivery_tag,
               'message': body}
    try:
        res = requests.post(FLASK_ENDPOINT, json=message,
                            timeout=1.)
    except (ReadTimeout, ConnectionError):
        print('Failed to connect to %s.' % FLASK_ENDPOINT)
        # need to implement a retry here
        return

    if res.status_code == 200:
        print('Message forwarded to Flask')
        channel.basic_ack(delivery_tag=method_frame.delivery_tag)

connection = pika.BlockingConnection()
channel = connection.channel()
channel.basic_consume(on_message, queue='race')
try:
    channel.start_consuming()
except KeyboardInterrupt:
    channel.stop_consuming()

connection.close()
```

当消息发送到队列时，这个脚本会在 Flask 上执行 HTTP 调用。

> 还有一个 RabbitMQ 插件可将消息推送给 HTTP 端点，但如果想添加特定逻辑的代码，将这个桥接隔离到小脚本中提供了更多潜力。从可靠性和性能的角度看，这样做还可避免在 RabbitMQ 内部集成 HTTP 推送。

在 Flask 中，/event 端点可以是一个经典视图：

```
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/event', methods=['POST'])
def event_received():
```

```
    message = request.json['message']
    # do something...
    return jsonify({'status': 'OK'})
if __name__ == '__main__':
    app.run()
```

5.2.3 发布/订阅模式

前一个模式用职程来处理特定主题的消息，职程处理的消息完全来自队列。我们甚至添加了代码以确认消息是否被处理。

当希望将消息发送到多个职程时，可使用发布/订阅模式。

这种模式是构建通用事件处理系统的基础，实现维度很像前面提到的只有一个交换器和多个队列的模式。不同之处是交换器部分有一个 fanout 类型。

在该设置中，每个绑定到 fanout 交换器的队列将收到相同的消息。

通过 pubsub 模式，可根据需要将消息广播给所有微服务。

5.2.4 AMQP 上的 RPC

AMQP 也实现了一个同步的请求/响应模式，这意味着可使用 RabbitMQ 替换 HTTP JSON 调用，直接进行服务间交互。

这种模式非常吸引人的地方在于，可让两个微服务直接交互。有些框架，如 Nameko(http://nameko.readthedocs.io)就使用这种方式来构建微服务。

但对比于基于HTTP的REST或RPC方式，基于AMQP的RPC方式在使用上优点并不明显，除非打算设定特定的通信信道，而且这个通信信道可能不是公开的API。坚持单一API可让微服务保持简洁。

5.2.5 异步总结

本节介绍关于异步的下述要点：

- 每当微服务执行一些额外任务时，都应该使用异步调用。如果应用所做的事情和响应没有关系，就没理由阻塞请求。
- Celery 是执行后台进程的好方法。
- 服务之间的通信并不总是局限于任务队列。
- 发送事件是防止服务相互依赖的好方法。

- 可使用诸如 RabbitMQ 的消息代理来构建一个完备的事件系统，让微服务通过消息进行交互。
- Pika 可用来协调所有消息传递。

5.3　测试服务间交互

第 3 章曾提到，当编写一个服务调用另一个服务的功能测试时，最大的挑战是隔离所有网络调用。

本节将介绍如何模拟一个用请求构建的同步调用，以及如何模拟一个异步调用，并将异步调用提供给 Celery 职程和其他异步进程。

5.3.1　模拟同步调用

如果你使用 request 库执行所有调用(或使用一个基于 Request 且没有太多定制的库)，通过使用本章前及的传输适配器，独立工作就很容易实现。

requests-mock 项目(https://requests-mock.readthedocs.io)实现了一个适配器，这个适配器能在测试中模拟网络调用。

本章前面介绍了一个 Flask 应用示例，有一个 HTTP 端点通过地址/api 提供内容服务。

这个应用使用setup_connector()函数构建的请求会话，然后在视图中使用 get_connector()函数获取会话。

在下面的测试中，先通过 requests_mock.Adapter()获取适配器实例，然后调用 session.mount()将 requests_mock 的适配器挂载到会话中。

```
import json
import unittest
from flask_application import app, get_connector
from flask_webtest import TestApp
import requests_mock

class TestAPI(unittest.TestCase):
    def setUp(self):
        self.app = TestApp(app)
        # mocking the request calls
        session = get_connector(app)
        self.adapter = requests_mock.Adapter()
```

```
        session.mount('http://', self.adapter)

    def test_api(self):
        mocked_value = json.dumps({'some': 'data'})
        self.adapter.register_uri('GET', 'http://127.0.0.1:5000
                                  /api', text=mocked_value)
        res = self.app.get('/api')
        self.assertEqual(res.json['result']['some'], 'data')
```

使用适配器提供了手工注册响应的能力；对于远端服务(http://127.0.0.1:5000/api)上的端点，通过 register_uri 来手工注册响应。适配器会拦截这个调用，然后立刻返回一个模拟值。

在 test_api()测试中，先尝试访问应用视图，当需要调用外部服务时会确保使用提供的 JSON 数据。

requests-mock 使用正则表达式匹配 request，所以这是一个在测试中非常有用的适配器，能避免运行测试时的网络依赖问题。

模拟其他服务的响应需要完成大量工作且很难维护。这意味着还需要关注其他服务是如何变化的，所以当模拟行为不再是真实 API 的行为时，测试中的模拟也就失去意义了。

使用模拟能帮助构建良好的功能测试覆盖率，但你务必完成很好的集成测试。集成测试是在真实调用其他服务的部署环境中，针对服务进行的测试。

5.3.2 模拟异步调用

如果应用异步地发送或接收请求，设置一些测试要比同步调用更难一些。

异步调用意味着应用发送一些消息给某处，并不期望立刻返回结果(或者干脆忘掉它)。

异步调用还意味着应用可能对发送给它的事件做出反应，就像在使用 Pika 时所看到的几个模式一样。

1. 模拟 Celery

如果为 Celery 职程构建测试，运行测试最简单的方法是使用一个真实的 Redis 服务器。可轻松地在任何平台上运行 Redis 服务器。甚至 Travis-CI 都可运行一个。所以，不同于添加很多工作来模拟交互，Flask 代码将与 Redis 一起工作，真实地将任务发送给职程。

使用一个真实代理意味着可在测试中运行 Celery 职程，而这一切只是为了验证应

用是否发送了适当的任务格式。Celery 提供了一个 pytest 测试 fixture，可运行一个独立线程，并在测试结束时关闭它。

要使用 Redis 并指向测试，可用 fixture 配置 Celery。第一步是在包含 Celery 任务的测试目录下，创建一个 tasks.py 文件。

下面是这个文件的一个例子。请注意我们并没有创建一个 Celery 实例，但使用 @shared_task 装饰器来标记函数是 celery 任务。

```
from celery import shared_task
import unittest

@shared_task(bind=True, name='echo')
def echo(app, msg):
    return msg
```

这个模块实现了一个叫做 echo 的 Celery 任务，将回显字符串。为配置 pytest 来使用它，需要实现 celery_config 和 celery_ includes 的 fixture：

```
import pytest

@pytest.fixture(scope='session')
def celery_config():
    return {
        'broker_url': 'redis://localhost:6379',
        'result_backend': 'redis://localhost:6379'
    }

@pytest.fixture(scope='session')
def celery_includes():
    return ['myproject.tests.tasks']
```

celery_config 函数用来传递创建 Celery 职程需要的所有参数，celery_includes 仅导入要返回的模块列表。在本例中，将在 Celery 任务注册器中注册 echo 任务。

这里，测试可使用 echo 任务，有一个职程也会被真实地调用：

```
from celery.execute import send_task

class TestCelery(unittest.TestCase):
    @pytest.fixture(autouse=True)
    def init_worker(self, celery_worker):
        self.worker = celery_worker
```

```
    def test_api(self):
        async_result = send_task('echo', ['yeah'], {})
        self.assertEqual(async_result.get(), 'yeah')
```

这里注意，我们使用 send_task()来触发任务的执行。只要任务具有唯一名称，任何注册在 Celery 代理上的任务都可用这个函数来运行。

对所有任务进行命名，并在所有微服务中确保唯一性是最佳实践。

这样做的原因是，当一个微服务想要从职程运行一个任务时，不必为获得任务而导入该职程的代码。

在下面的代码示例中，echo 任务在独立的微服务中运行，只需要知道任务名(而不必导入代码)，就可通过 send_task()调用来触发任务；每个交互都通过 Redis 进行：

```
>>> import celery
>>> redis = 'redis://localhost:6379'
>>> app = Celery(__name__, backend=redis, broker=redis)
>>> f = app.send_task('echo', ['meh'])
>>> f.get()
'meh'
```

回到你的测试，如果测试模拟了一些 Celery 职程，请确保实现了任务的远端应用上每个任务都有一个名称，还要确保正在测试的应用在代码中使用 send_task()调用任务。

通过这种方式，Celery fixture 会神奇地为应用模拟职程。

最后，应用不会同步地等待 Celery 职程返回结果，所以在 API 调用完成后，需要对测试用的职程进行检查。

2. 模拟其他异步调用

如果你使用 Pika 和 RabbitMQ 进行消息处理，Pika 库会直接使用 socket 模块和服务器交互，这让模拟变得很痛苦，因为我们要在线路上跟踪数据是如何发送和接收的。

像 Celery 一样，可为测试运行一个本地 RabbitMQ 服务器(Travis-CI 也提供了这个功能，可参见 https://docs.travis-ci.com/user/database-setup/)。

这时，发送消息就和平时一样了，可创建一个脚本从 Rabbit 队列中捡起消息并进行验证。

当需要测试从 RabbitMQ 收到事件的进程时，如果是通过 HTTP 调用发生的(就像我们在 AMQP-to-HTTP 的桥接中所做的一样)，可简单地通过手工触发事件进行测试。

最重要的是在运行测试时确保不依赖其他微服务。但是，只要能在专有的测试环

境运行它们，对诸如 Redis 或 RabbitMQ 的消息服务器的依赖就不是问题。

5.4　本章小结

本章介绍了一个服务如何通过使用请求会话，与其他服务同步地交互；还介绍了一个服务如何通过使用 Celery 职程或更高级的 RabbitMQ 消息模式，与其他服务异步地交互。

通过模拟其他服务(但并不需要模拟消息代理)，可独立地测试服务。

独立地测试每个服务非常有用，不过当错误发生时，很难知道发生了什么，当 bug 发生在一系列异步调用中时尤其如此。

这种情况下，通过中心化的日志系统来跟踪发生了什么将很有帮助。第 6 章将说明如何利用微服务来跟踪它们的活动。

第 6 章

监 控 服 务

在前一章中，已测试了彼此交互的独立服务。但当真实的部署环境发生问题时，需要对当前情况有一个全局性认识。例如，当第一个微服务调用第二个微服务，而第二个微服务又调用第三个微服务时，就很难理解到底是哪一个出了问题。我们需要能够跟踪用户与引发问题的系统的所有交互信息。

Python 应用生成的日志可帮助调试问题，但在不同服务器之间收集所需的信息会非常困难。幸好，可通过集中化日志来监控一个分布式部署环境。

持续监控服务对于确定整个系统的健康状况和跟踪各个部分的运转也很重要。这涉及一些问题，例如，是否有一个服务正在危险地接近占用 100%的内存？某个微服务每分钟处理多少请求？是不是为某个 API 部署了太多服务器，能否移除部分实例以减少费用？刚部署的应用是否对性能产生了负面影响？

为持续地回答这些问题，每个部署的微服务都需要有一种机制，能向监控系统报告主要指标。

本章主要由以下两部分组成：

- 集中化日志
- 性能指标

在本章结束后，你将对如何设置微服务并监控它们有一个完整的认识。

6.1 集中化日志

Python内置了logging 包，它能将日志以流的形式输出到不同地方，包含标准输出、轮转日志文件、syslog、TCP或UDP套接字。

Flask 应用甚至能将日志输出到 SMTP 服务器。在下例中，当 email_errors 装饰的函数抛出异常时，email_errors 装饰器会发送一份邮件。注意这个处理器会使用 telnet 会话连接 SMTP 服务器来发送邮件。如果这个会话出现问题，就可能在调用 logger.exception()函数时抛出第二个异常：

```
import logging
from logging.handlers import SMTPHandler

host = "smtp.example.com", 25
handler = SMTPHandler(mailhost=host, fromaddr="tarek@ziade.org",
                      toaddrs=["tarek@ziade.org"],
                      subject="Service Exception")

logger = logging.getLogger('theapp')
logger.setLevel(logging.INFO)
logger.addHandler(handler)

def email_errors(func):
   def _email_errors(*args, **kw):
      try:
         return func(*args, **kw)
      except Exception:
         logger.exception('A problem has occured')
         raise
return _email_errors

@email_errors
def function_that_raises():
   print(i_dont_exist)

function_that_raises()
```

如果上面的代码能正常调用，就能收到一封包含完整异常回溯信息的电子邮件。



Python的日志包中有许多内置的处理器，见https://docs.python.org/3/library/logging.handlers.html。

在开发服务时，将日志输出到标准输出或日志文件是合适的，但如前文所述，在分布式系统中，这种方式无法扩展。

用邮件发送错误信息是一种改进，但高流量的微服务可能在一小时内产生上千个重复的异常信息——意味着发送上千封重复邮件。同时，如果滥发大量邮件，那么服务所在的服务器地址可能被收件箱的 SMTP 服务器列入黑名单，服务也会因为忙于发送大量邮件而无法响应请求。

因此，分布式系统需要更好的办法来处理错误信息，例如用足够小的开销来收集所有微服务的日志，同时提供一些可视化的用户界面。

目前已有一些系统能集中处理 Python 应用生成的日志。大部分系统可接收 HTTP 或 UDP 协议的负载。通常优先选择后者，因为后者可减少应用发送数据时的开销。

Python 社区中有一个著名的工具叫 Sentry(https://sentry.io/)，它可集中管理错误日志，还提供美观的用户界面来展示回溯信息。Sentry 能在服务发生问题时检测和聚合错误。它的用户界面提供了简明的解决方案工作流程，让人们处理问题。

但 Sentry 专注于错误处理，并不适合通用日志。如果想得到错误之外的日志，就得另寻他路。

另一个开源方案是 Graylog(http://graylog.org)。它是一个通用日志应用，提供基于 Elasticsearch 的强大搜索引擎(日志保存在 Elasticsearch 中)，使用 MongoDB(https://www.mongodb.com/)存储应用数据。

Graylog 支持自定义的日志格式和其他备选格式(如简单 JSON 格式)，因此可接受任何日志。它包含一个内置的日志收集器，也可在配置后与其他收集器(如 fluentd)一起工作。

6.1.1 设置 Graylog

Graylog 服务器是一个 Java 应用，它使用 MongoDB 作为数据库，将收到的所有日志保存在 Elasticsearch 中。Graylog 栈有很多难以设置和管理的移动部件，如果要自行部署，那么需要专门人员来实施。

生产环境的典型配置是：一个专有的 Elasticsearch 集群和若干个 Graylog 节点，每个节点都有一个 MongoDB 实例。更多信息可参阅 Graylog 架构文档(http://docs.graylog.org/en/latest/pages/architecture.html)。

使用 Docker(https://docs.docker.com)镜像是一个不错的尝试 Graylog 的方式，有关它的详细说明见 http://docs.graylog.org/en/latest/pages/installation/docker.html。



第 10 章解释如何使用 Docker 来部署微服务，并介绍构建和运行 Docker 镜像所需的基本知识。

与 Sentry 类似，Graylog 是一个商业公司支持的项目，这个公司提供不同的托管

方案。根据项目的类型和体量，避免自行维护 Graylog 的基础设施可能是一个好方案。例如，如果要运行一个有服务级别协议(Service-Level Agreement，SLA)的商业项目，顺利运行一个 Elasticsearch 集群不是一项小任务，需要引起注意。

但对那些不会产生大量日志的项目，或日志管理的短暂宕机并不意味着世界末日情况下，自行维护 Graylog 栈也可能是个好方案。

本章仅使用 Docker 镜像、docker-compose(通过一个调用就能运行和绑定多个 docker 镜像的工具)和最少的设置来演示微服务如何与 Graylog 交互。

为在本地运行 Graylog 服务，需要确保已安装 Docker(见第 10 章)，然后使用下面的 Docker compose 配置(来自 Graylog 文档)：

```
version: '2'
services:
  some-mongo:
    image: "mongo:3"
  some-elasticsearch:
    image: "elasticsearch:2"
    command: "elasticsearch -Des.cluster.name='graylog'"
  graylog:
    image: graylog2/server:2.1.1-1
    environment:
      GRAYLOG_PASSWORD_SECRET: somepasswordpepper
      GRAYLOG_ROOT_PASSWORD_SHA2:
8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918
      GRAYLOG_WEB_ENDPOINT_URI: http://127.0.0.1:9000/api
    links:
      - some-mongo:mongo
      - some-elasticsearch:elasticsearch
    ports:
      - "9000:9000"
      - "12201/udp:12201/udp"
```

将这个文件以文件名 docker-compose.yml 保存，之后在包含它的目录中运行 docker-compose up，这样 Docker 就会拉取 MongoDB、Elasticsearch 和 Graylog 镜像，然后运行它们。

一旦运行，就能在浏览器中打开 http://localhost:9000 来查看 Graylog 仪表盘，然后使用 admin 作为用户名和密码来使用它。

接着通过 System | Inputs 命令添加一个 UDP 输入，以便 Graylog 接收微服务的日志。

为此，在端口 12012 创建一个新的 GELF UDP 输入，如图 6-1 所示。

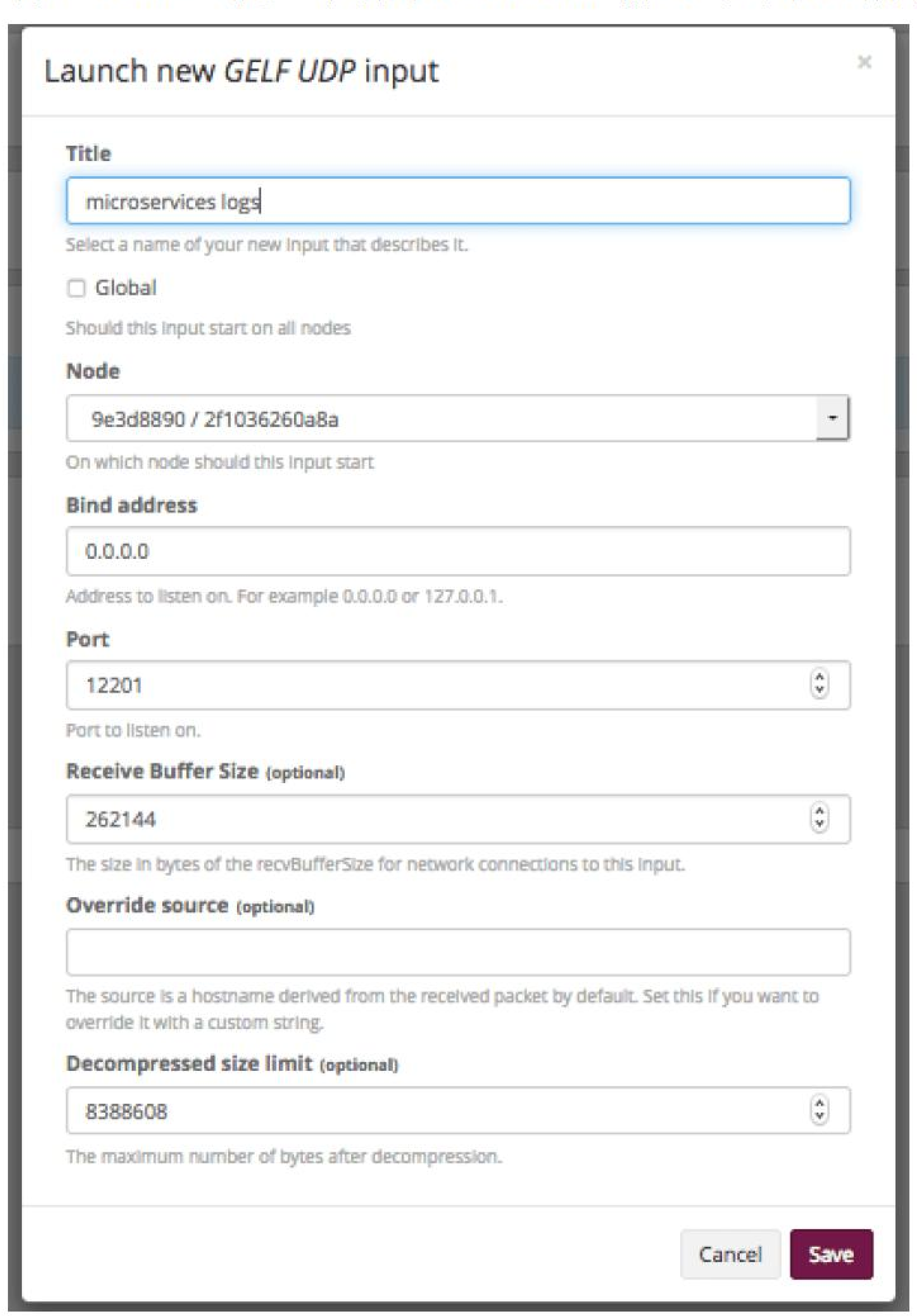


图 6-1 创建新的 GELF UDP 输入

一旦新的输入就绪，Graylog 就能绑定 UDP 端口 12201，然后准备好接收数据。docker-compose.yml 文件已经公开了 Graylog 镜像的这个端口，所以 Flask 应用可通过本机发送数据。

如果在新输入上点击 Show Received Messages，就能得到包含了所有收集到的日志的结果，如图 6-2 所示。

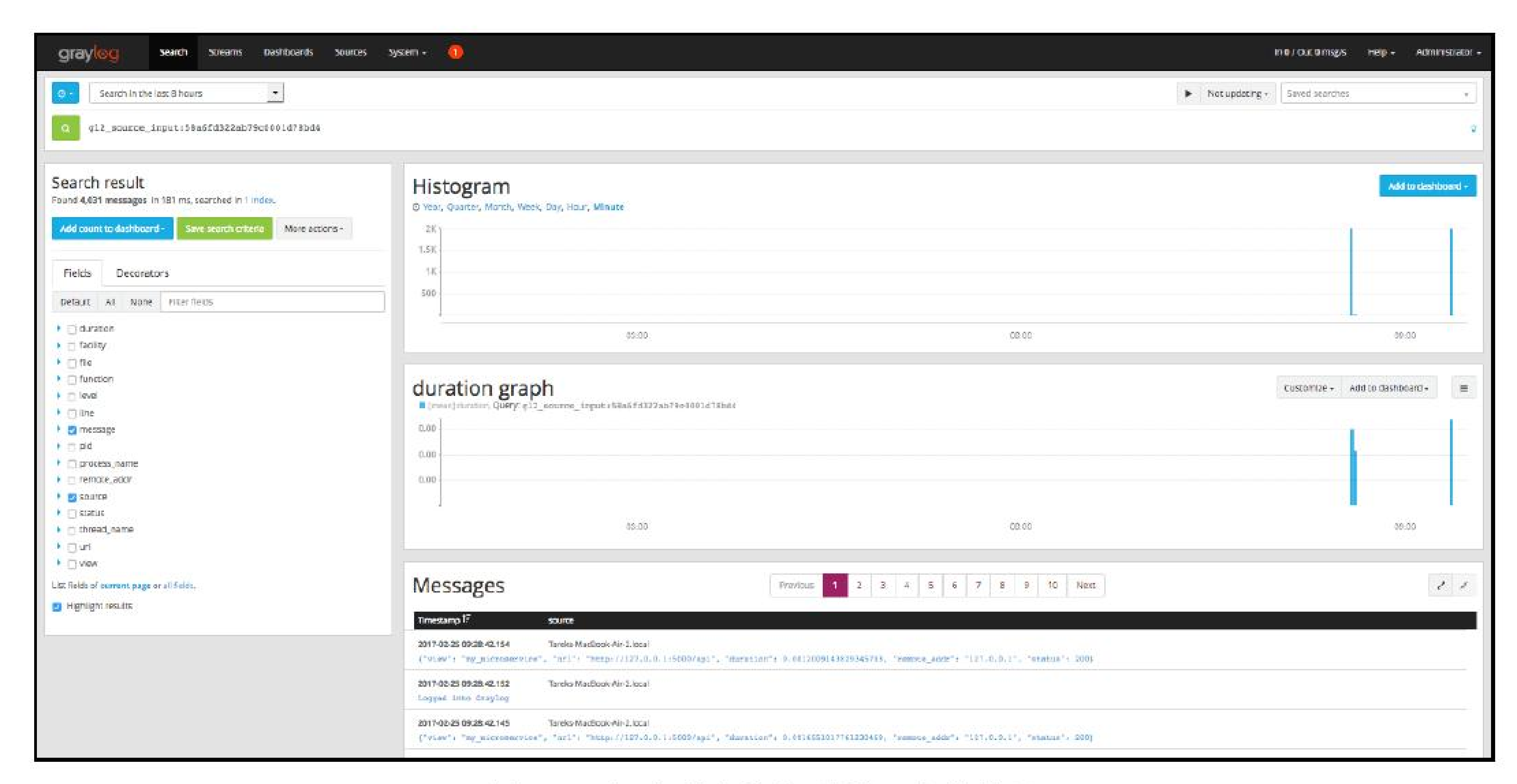


图 6-2 包含所有收集到的日志的结果

恭喜！现在已经准备好在一个集中位置接收日志，然后在 Graylog 仪表盘上在线查看它们。

6.1.2 向 Graylog 发送日志

为从 Python 向 Graylog 发送日志，可使用 Graypy(https://github.com/severb/graypy)。它将 Python 日志转换成 Graylog 扩展日志格式(GELF，详见 http://docs.graylog.org/en/latest/pages/gelf.html)。

Graypy 默认通过 UDP 发送日志，但是，如果需要百分之百确保日志都能发送到 Graylog，那么可通过 AMQP 发送日志。



大多数情况下，使用 UDP 来集中化日志已经够用。但与 TCP 不同，使用 UDP 时可能丢弃一些数据包，且无法发现。如果你的日志策略中需要更多保证，那么基于 RabbitMQ 的协议会更可靠。

为使用 Graypy，需要将内置的日志处理器替换成它提供的处理器：

```
handler = graypy.GELFHandler('localhost', 12201)
logger = logging.getLogger('theapp')
logger.setLevel(logging.INFO)
logger.addHandler(handler)
```

graypy.GELFHandler 类会将日志转换成 UDP 负载，然后发给 GELF UDP 输入。上面的例子中，这个输入会监听本机的 12201 端口。

发送 UDP 负载的代码不太可能引发错误，开销也最低，因为不会确认对方已经读取了 UDP 数据报。

为将 Graypy 集成到 Flask 应用中，可直接在 app.logger 上添加处理器。也可在每次 Flask 因为异常(无论是有意还是无意的)而终止处理请求时，在已注册的错误处理程序中自动记录异常。

```
import logging
import graypy
import json
from flask import Flask, jsonify
from werkzeug.exceptions import HTTPException, default_exceptions

app = Flask(__name__)

def error_handling(error):
    if isinstance(error, HTTPException):
        result = { 'code': error.code, 'description':
                   error.description}
    else:
        description = default_exceptions[500].description
        result = {'code': 500, 'description': description}

    app.logger.exception(str(error), extra=result)
    result['message'] = str(error)
    resp = jsonify(result)
    resp.status_code = result['code']
    return resp


for code in default_exceptions.keys():
    app.register_error_handler(code, error_handling)

@app.route('/api', methods=['GET', 'POST'])
def my_microservice():
    app.logger.info("Logged into Graylog")
    resp = jsonify({'result': 'OK', 'Hello': 'World!'})
    # this will also be logged
    raise Exception('BAHM')
    return resp
```

```
if __name__ == '__main__':
    handler = graypy.GELFHandler('localhost', 12201)
    app.logger.addHandler(handler)
    app.run()
```

调用/api 时，应用会给 Graylog 发送简单日志，以及包含完整回溯的异常。图 6-3 展示了这个例子生成的回溯信息。

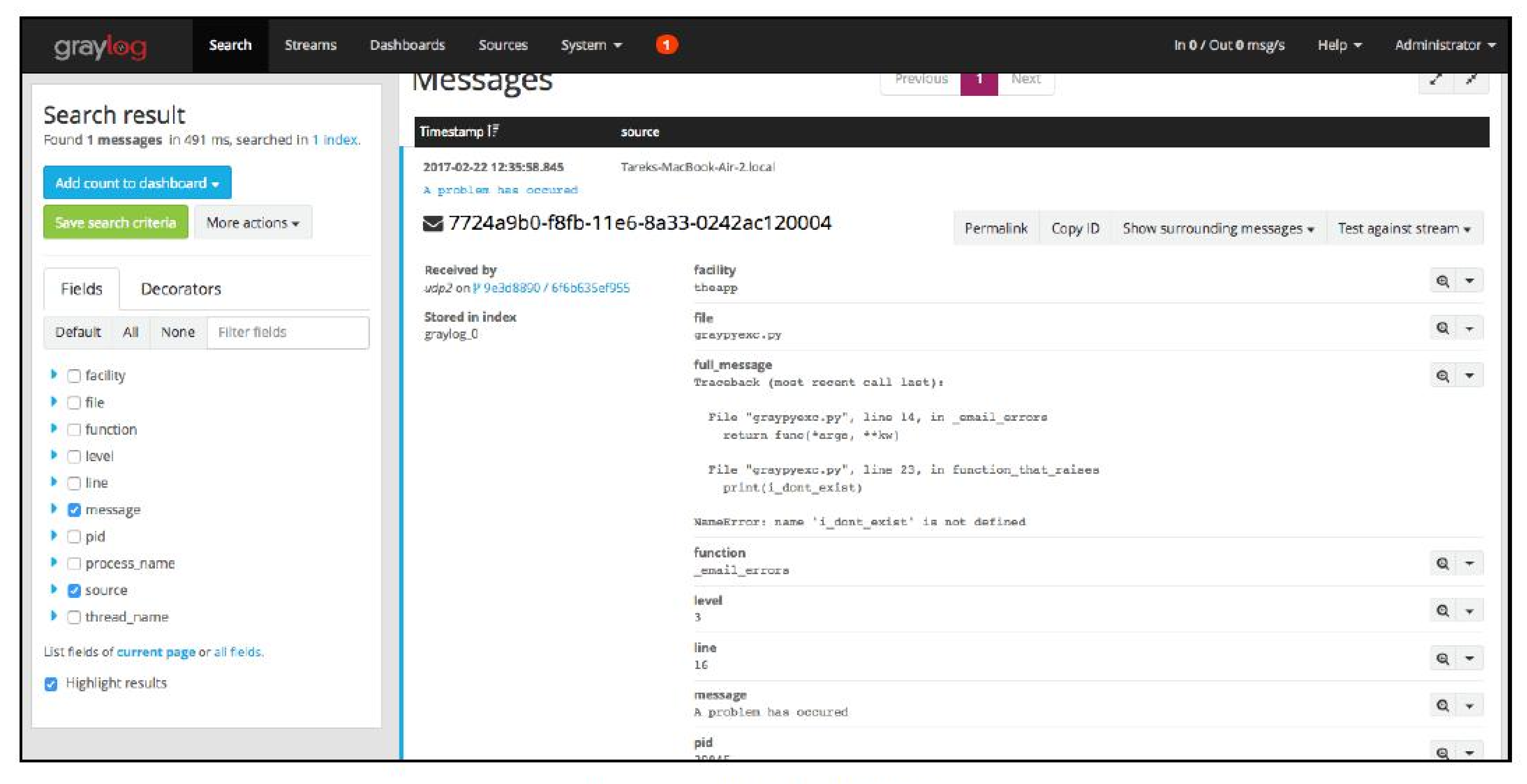


图 6-3 生成的回溯信息

用户也会在 JSON 响应中收到这个错误。

6.1.3 添加扩展字段

Graypy 给每条日志添加一些元数据字段，例如：

- 远程机器的地址
- PID(进程 ID)、进程和线程名
- 进行调用的函数名

Graylog 自身会添加接收日志的主机名(作为 source 字段)和其他一些字段。

对于分布式系统，需要添加更多上下文信息以便高效地搜索日志。例如，当需要在同一个用户会话中搜索一系列调用时，用户名特别有用。

这个信息通常存储在微服务的 app.session 中。可使用 logging.Filter 类，将它添加到发送给 Graylog 的每个日志记录中。

```
from flask import session
import logging

class InfoFilter(logging.Filter):
    def filter(self, record):
        record.username = session.get('username', 'Anonymous')
        return True

app.logger.addFilter(InfoFilter())
```

添加这个过滤器后，会将 username 字段添加到每个 Graylog 条目中。

你能想到的有助于理解正在发生的事情的任何上下文信息，都应该发送到 Graylog 中。尽管如此，还需要注意，在日志中添加更多数据时会有一些负面效果。如果日志包含太多细节，搜索会变得低效，尤其在一个请求会产生多条日志的情况下。

前面介绍了微服务如何通过 UDP 以最小开销将所有日志发送到一个集中化服务中。一旦存储日志，这个集中化服务应提供有效的搜索功能。

保留所有日志，在调研微服务的问题时会非常有用——但希望这种情况不会经常发生。

另一方面，能持续监控应用和服务器性能，将能让你在其中一台服务器瘫痪时保持主动。



Graylog 企业版是一个托管版本的 Graylog，它包含一些额外功能，如归档旧日志，详见 https://www.graylog.org/enterprise/feature/archiving。

6.2　性能指标

当微服务占用全部内存时，会发生糟糕的事情。一些 Linux 发行版会使用臭名昭著的“内存杀手”(out-of-memory killer，oomkiller)来杀掉贪婪的进程。

以下原因可能导致使用过多内存：

- 微服务存在内存泄漏，且内存占用量稳定增长，有时增长非常快。Python 的 C 扩展中常忘记解除对象的引用，然后在每次调用时泄漏内存。
- 代码在使用内存时无所顾忌。例如，用作临时内存缓存的字典会在几天内无限增长——除非设计上有限制。
- 分配给服务的内存不足——服务器收到太多请求，或处理任务时太吃力。

能跟踪内存使用随时间的变化情况，并在用户受到影响前了解这些问题是很重

要的。

在生产环境中接近 100%的 CPU 使用率也会有问题。尽管最大化 CPU 使用率是很值得的，但如果服务器太忙，当有新请求到来时，服务将无法响应。

最后，如果知道服务器磁盘几乎已满，可采取措施防止服务因空间不足而崩溃。

但愿在项目进入生产环境前，可通过负载测试发现大部分问题。负载测试是一个确定服务器在测试期间和一段时间内能承受多少负载，以及根据预期负载来调用 CPU 和内存资源的好方法。

为此，使用服务来持续监控系统资源。

6.2.1 系统指标

基于 Linux 的系统让监控 CPU、内存和磁盘变得简单。有一些持续更新的系统文件包含这些信息，还有很多工具可读取它们。诸如 top 的命令行工具能跟踪所有运行的进程，并根据内存或 CPU 排序。

Python 中的 psutil(https://pythonhosted.org/psutil)项目是一个跨平台的库，利用它能以编程方式获取这些信息。

结合 graypy 包，可编写一个小脚本，将系统指标持续发送给 Graylog。

以下示例中，asyncio 循环将每秒的 CPU 使用百分比发送给 Graylog：

```
import psutil
import asyncio
import signal
import graypy
import logging
import json

loop = asyncio.get_event_loop()
logger = logging.getLogger('sysmetrics')
def _exit():
    loop.stop()

def _probe():
    info = {'cpu_percent': psutil.cpu_percent(interval=None)}
    logger.info(json.dumps(info))
    loop.call_later(1., _probe)

loop.add_signal_handler(signal.SIGINT, _exit)
```

```
loop.add_signal_handler(signal.SIGTERM, _exit)
handler = graypy.GELFHandler('localhost', 12201)
logger.addHandler(handler)
logger.setLevel(logging.INFO)
loop.call_later(1., _probe)

try:
    loop.run_forever()
finally:
    loop.close()
```

以守护进程方式在服务器上运行这段脚本，就能跟踪 CPU 的使用情况。

System-metrics(https://github.com/tarekziade/system-metrics/)项目的脚本与此基本相同，但添加了内存、磁盘和网络信息。如果使用 pip install 命令，那么可通过一个命令行脚本来探测系统。

一旦脚本运行，就能在 Graylog Web 应用中创建一个包含一些小组件的仪表盘(见 http://docs.graylog.org/en/latest/pages/dashboards.html)。还可创建一个警报，用于在特定情况下发送警报。可在 Graylog 的 stream 中创建警报，它会实时处理传入的消息。

为在 CPU 使用率超过 70%时发送邮件，可创建一个 stream，然后使用 stream 规则来收集 psutil 脚本发送的 cpu_percent 字段，如图 6-4 所示。

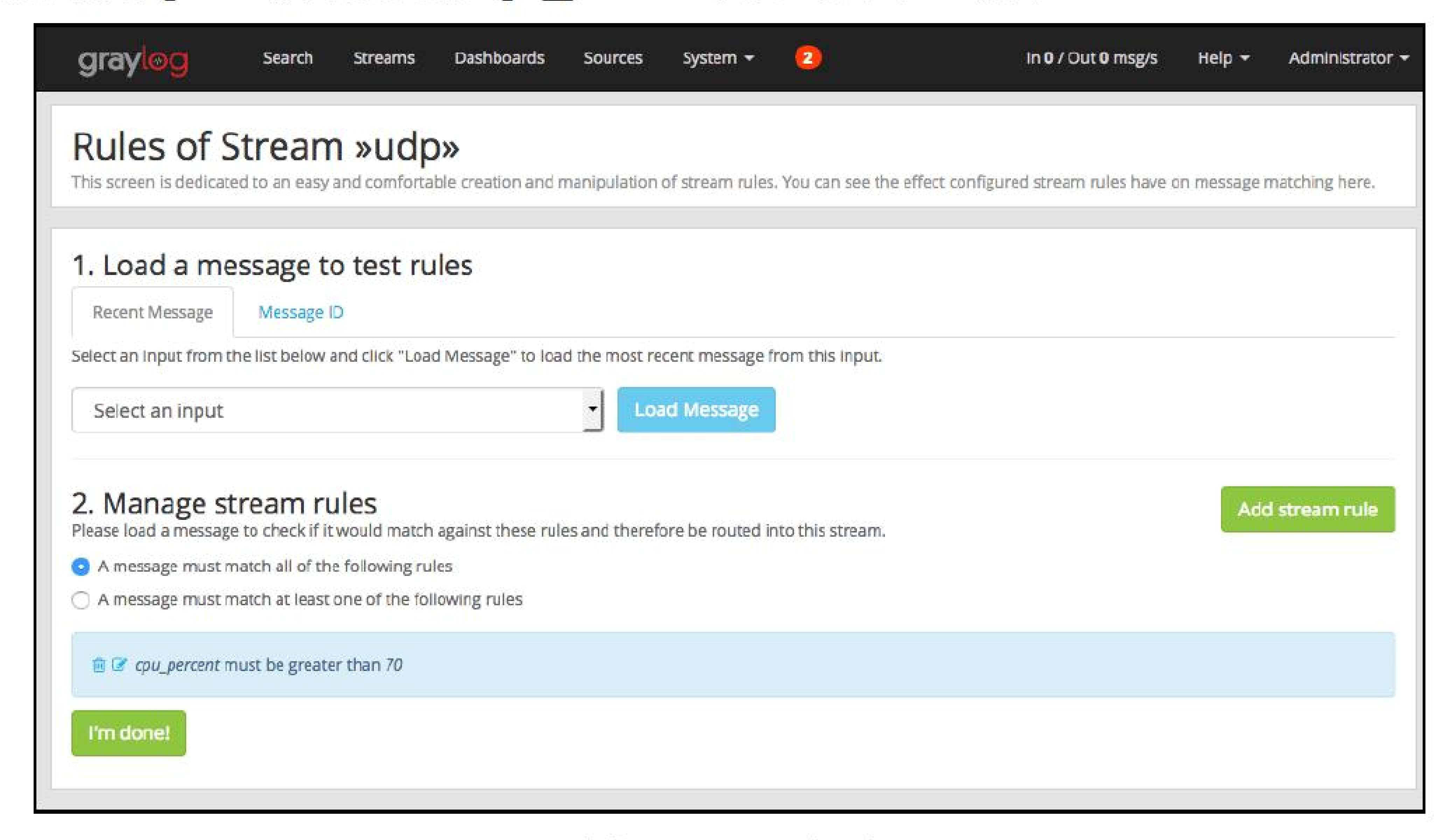


图 6-4 stream 规则

至此，可管理 stream 的警报，然后添加一条 email 警报，使其在满足条件一定时间后触发。

6.2.2 代码指标

对一些微服务而言，能得到代码中的一些性能指标会很有用。

例如 New Relic 通过封装 Flask 内部的一些调用来跟踪 Jinja2 和数据库调用的性能，以测量生成模板或执行数据库调用的时长。

但如果要在代码中添加测量工具，并将它们部署到生成环境，那么需要非常小心。稍不留意会减慢服务。例如，使用 Python 内置的分析器是无法想象的，因为它增加了相当多的开销。

一种简单模式是显式地指定那些需要测量的函数。

下面的例子中，@timeit 装饰器会收集 fast_stuff()和 some_slow_stuff()两个函数的执行时间，然后在每个请求结束时给 Graylog 发送一条包含时长的消息：

```
import functools
import logging
import graypy
import json
import time
import random
from collections import defaultdict, deque
from flask import Flask, jsonify, g

app = Flask(__name__)

class Encoder(json.JSONEncoder):
    def default(self, obj):
        base = super(Encoder, self).default
        # specific encoder for the timed functions
        if isinstance(obj, deque):
            calls = list(obj)
            return {'num_calls': len(calls), 'min': min(calls),
                    'max': max(calls), 'values': calls}
        return base(obj)

def timeit(func):
    @functools.wraps(func)
    def _timeit(*args, **kw):
        start = time.time()
        try:
```

```
            return func(*args, **kw)
        finally:
            if 'timers' not in g:
                g.timers = defaultdict(functools.partial(deque,
maxlen=5))
            g.timers[func.__name__].append(time.time() - start)
    return _timeit

@timeit
def fast_stuff():
    time.sleep(.001)

@timeit
def some_slow_stuff():
    time.sleep(random.randint(1, 100) / 100.)

def set_view_metrics(view_func):
    @functools.wraps(view_func)
    def _set_view_metrics(*args, **kw):
        try:
            return view_func(*args, **kw)
        finally:
            app.logger.info(json.dumps(dict(g.timers), cls=Encoder))
    return _set_view_metrics

def set_app_metrics(app):
    for endpoint, func in app.view_functions.items():
        app.view_functions[endpoint] = set_view_metrics(func)

@app.route('/api', methods=['GET', 'POST'])
def my_microservice():
    some_slow_stuff()
    for i in range(12):
        fast_stuff()
    resp = jsonify({'result': 'OK', 'Hello': 'World!'})
    fast_stuff()
    return resp
```

```
if __name__ == '__main__':
    handler = graypy.GELFHandler('localhost', 12201)
    app.logger.addHandler(handler)
    app.logger.setLevel(logging.INFO)
    set_app_metrics(app)
    app.run()
```

使用这些测量工具后，就能在 Graylog 中追踪每个调用的时长，如图 6-5 所示。

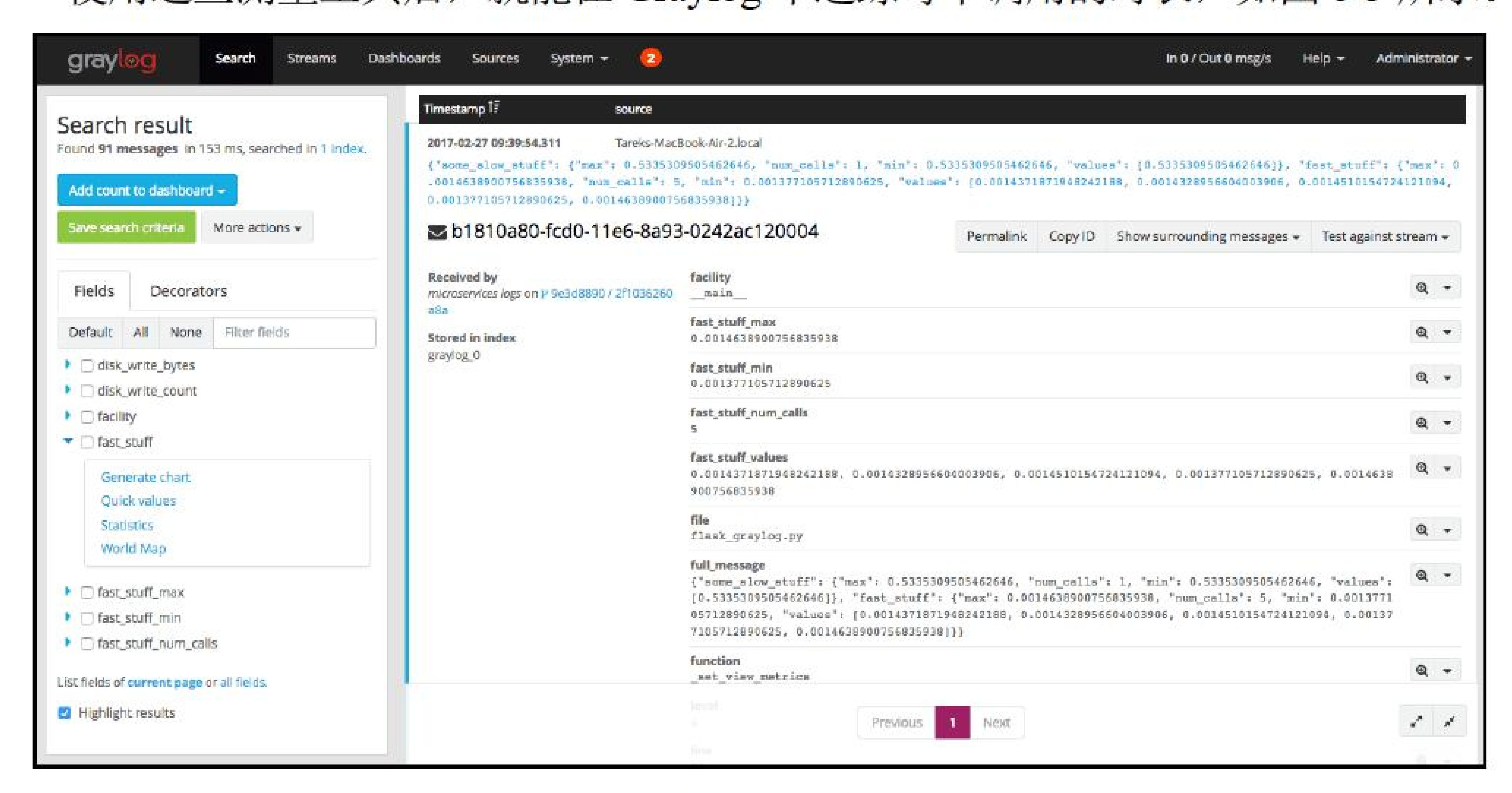


图 6-5　追踪每个调用的时长

6.2.3　Web 服务器指标

最后一个要在集中化日志中添加的指标与 HTTP 请求和响应相关。可在 Flask 应用内部与定时器一起添加这些指标，但更好的方式是在 Web 服务器级别实现，这样可减少开销，并让这些指标能兼容那些并非由 Flask 生成的内容。

例如，假设 nginx 直接支持静态文件，而我们还想跟踪服务的情况。Graylog 有一个集市(https://marketplace.graylog.org)来使用内容包(content pack)对其进行扩展。nginx 内容包(https://github.com/Graylog2/graylog-contentpack-nginx)能解析 nginx 的访问日志和错误日志，并将其推送到 Graylog。

这个内容包配备一个默认的仪表盘，它利用 nginx 的能力，使用 syslog 来发送 UDP 日志。

使用这个配置，就能跟踪一些有价值的信息，例如：

- 平均响应时间
- 每分钟的请求数量

- 远程地址
- 调用点和请求的方法
- 状态码和响应的大小

结合特定于应用的指标和系统指标，所有这些日志都可用于构建实时仪表盘，用来跟踪部署环境中正在发生的情况，如图 6-6 所示。

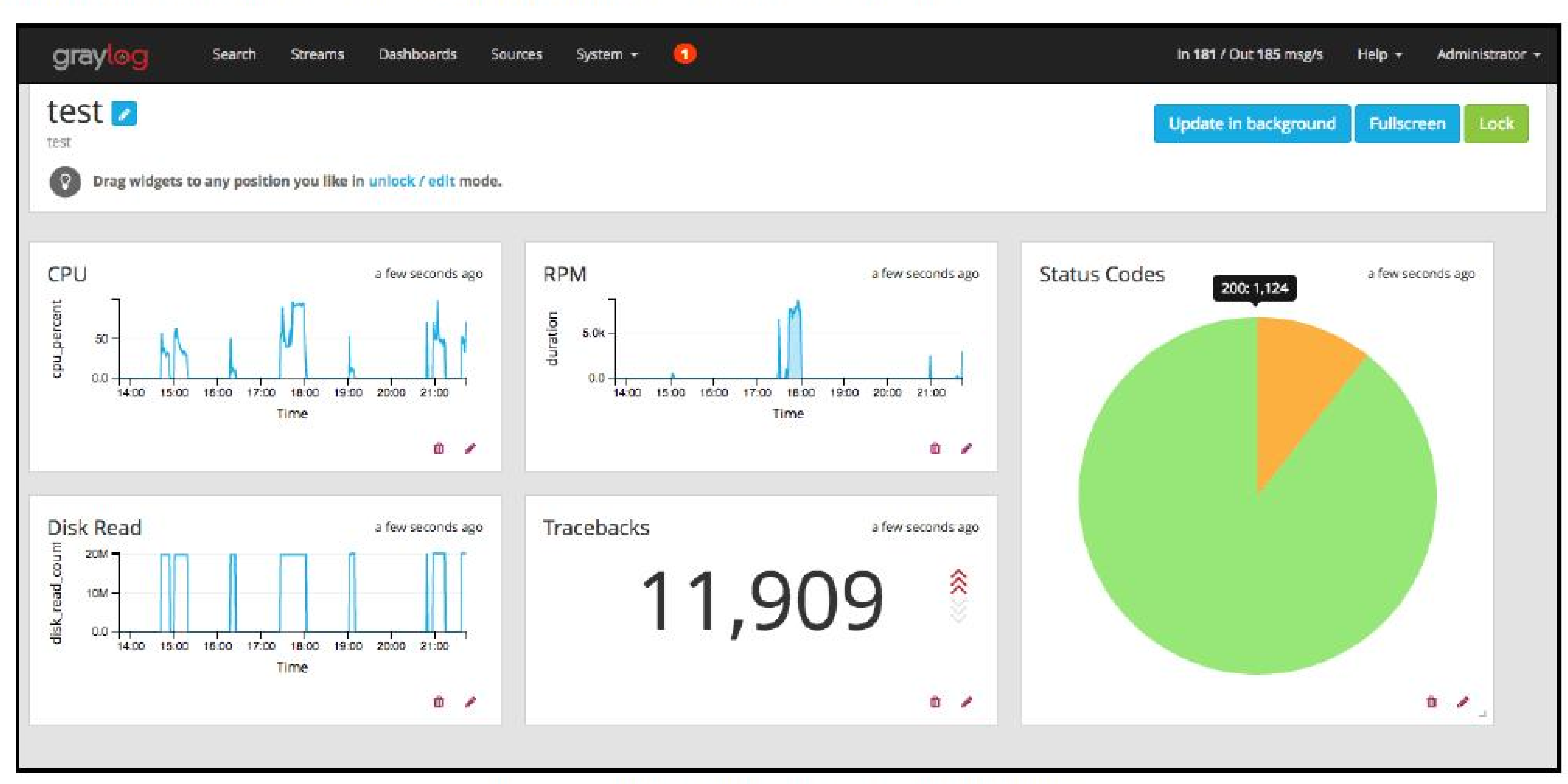


图 6-6　跟踪部署环境中正在发生的情况

6.3　本章小结

本章介绍了如何在微服务和 Web 应用级别添加测量工具。还讲解了如何设置 Graylog 来集中管理日志并使用生成的日志与性能指标。

Graylogs 使用 Elasticsearch 来存储所有数据，这个选择提供出色的搜索功能，便于你查找正在发生的事情。添加警报的能力也很有用，出错时，就会收到通知。但部署 Graylog 时需要仔细考虑。一旦有了大量数据，运行和维护 Elasticsearch 集群的工作会变得繁重。

对于一些指标，基于时序的系统，例如来自 InfluxData(https://www.influxdata.com/)的 InfluxDB(开源)是快速和轻量的替代品。但这并不意味着存储原始的日志和异常。

如果很在意性能指标和异常，那么一个可能的良好防范措施是结合这些工具：使用 Sentry 来处理异常，使用 InfluxDB 来跟踪性能。无论如何，只要应用和 Web 服务器通过 UDP 生成了日志和指标，那么从一个应用切换到另一个工具会更简单。

第 7 章将关注微服务开发的另一个重要方面：保护 API，提供一些身份验证方案，以及避免欺骗和滥用。

第 7 章

保护服务

到目前为止，除了身份验证和授权，服务之间的交互都介绍完毕了。每个 HTTP 请求都乐观地返回结果。但这样的情况不会在生产环境中发生，有两个简单原因：

- 要知道谁正在请求服务(身份验证)；
- 要知道是否允许请求者发起请求(授权)。

例如，大部分情况下，都不会允许一个匿名的服务请求者删除数据库中的数据项。

在单体 Web 应用中，身份验证发生在登录页面，一旦用户身份被确认，身份信息就会写入 Cookie，并可用在随后的请求中。

在基于微服务的架构中，很难继续使用这个方案，因为微服务不是用户，无法使用 Web 页面进行身份验证。需要采用一种方式来自动允许或拒绝服务间的调用。

OAuth2 授权协议(https://oauth.net/2/)可在微服务上灵活地添加身份验证和授权，可用来验证用户和服务的身份。本章将介绍 OAuth2 的一些特性，并学习如何验证微服务的身份，这个微服务用来保护服务间的交互。

保护服务也意味着要避免对系统的各种欺诈和滥用。例如，如果一个客户端开始攻击一个端点，无论这个行为是恶意攻击，还只是由无敌意的 bug 触发，都需要检测这个行为并尝试保护系统。诚然，对于非常严重的 DDoS(Distributed Denial of Service)攻击，我们是无能为力的，但构建一个基本 Web 应用防火墙还是很容易的，也是保护系统免受常见攻击的好办法。

最后，还可通过代码级别的修改来保护微服务，如控制系统调用，或确保 HTTP 重定向的最终页面不是一个恶意网页。本章最后将列举一些代码级保护措施，并演示如何通过持续扫描代码找到潜在的安全问题。

以下是本章涵盖的主题：

- OAuth2 协议概述；

- 在实践中基于令牌(token)的身份验证是如何工作的；
- 什么是 JWT 标准，以及如何在 TokenDealer 服务中用它来保护微服务；
- 如何实现 Web 应用防火墙；
- 一些保护微服务代码的最佳实践。

7.1 OAuth2 协议

OAuth2 是一个广泛采用的标准，用来保护 Web 应用，以及保护应用与用户之间或与其他 Web 应用之间的交互。但因为采用了许多复杂的 RFC 技术，所以很难理解。

OAuth2 的核心思想是：一个中心化服务负责验证请求者的身份，并以代码或令牌方式授予一些访问权限，可称这些令牌或代码为钥匙。一旦提供资源的微服务接受了钥匙，用户或服务就可以使用这些钥匙来访问资源。

第 4 章就使用这种方式构建了 Strava 微服务。在通过 Strava 的身份验证微服务授予访问权限后，发起请求的服务就能采用和用户一样的方式请求 Strava API。这种授权方式称为授权代码许可(Authorization Code Grant，ACG)，这是最常用的授权方法，也称为三段式 OAuth，因为它涉及用户、身份验证微服务和第三方应用。Strava 生成一段可用于请求其 API 的授权代码，我们创建的 Strava Celery 职程在每次调用时都会使用它。

在图 7-1 中，典型场景是让用户和应用进行交互，应用将访问诸如 Strava 的微服务。当用户调用应用时①，会被重定向到 Strava 服务，然后 Strava 服务会授予应用访问 Strava API 的权限②，一旦完成，应用会从 HTTP 回调中获得授权代码，此后就能以用户身份来访问 Strava API③。

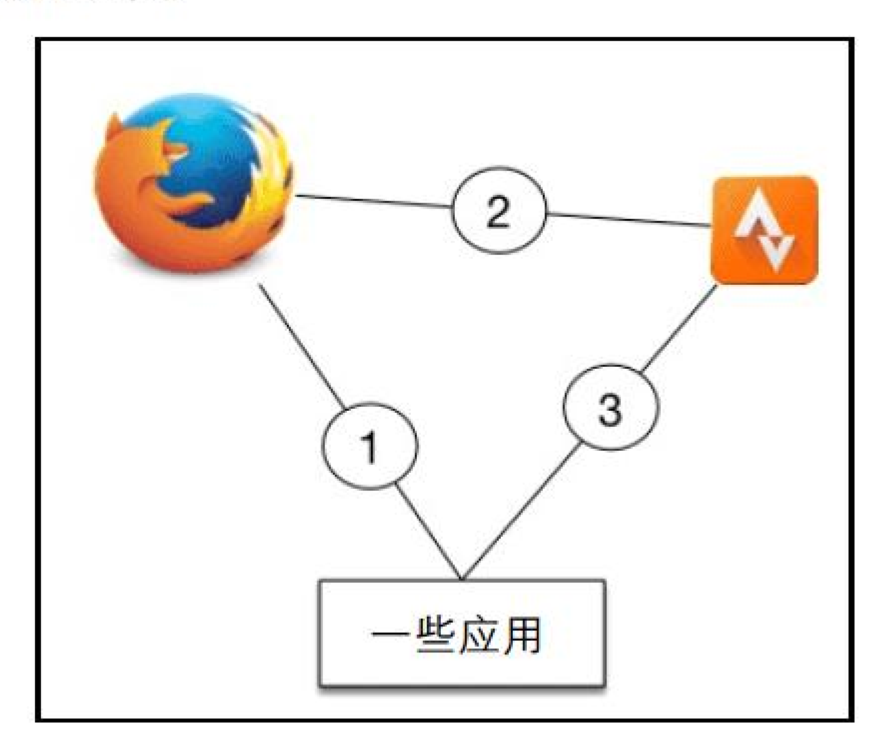


图 7-1　用户与应用交互

对于不是由特定用户发起的服务间身份验证，还有一种授权类型，称为客户端凭证授权(Client Credentials Grant，CCG)，服务 A 可向身份验证微服务请求验证，通过

验证后会获得一个用来访问微服务 B 的令牌。



更多相关信息，请参阅 OAuth2 Authorization Framework 4.4 节中描述的 CCG 方案(https://tools.ietf.org/html/rfc6749#section-4.4)。

这种方式和授权方式非常相似，但服务无法像用户那样跳转到 Web 页面。通过用密钥换取令牌的方式而隐式获得了授权。

对于基于微服务的架构，使用这两种授权方法能集中处理整个系统的身份验证和授权。通过构建实现了 OAuth2 协议的微服务作为身份验证服务，并跟踪服务之间是如何交互的，就得到一个可减少系统安全问题的解决方案，所有发起请求的微服务都通过构建一个实现了部分 OAuth2 协议的微服务进行身份验证，一切都集中在一起。

CCG 流是本章最有趣的部分，因为它能在不依赖于用户的情况下保护微服务之间的交互。还简化了权限管理，可根据上下文发布具有不同使用范围的令牌。

如果第三方以特定用户的身份向某些服务发起请求，则可添加三段式身份验证来应对这个场景。不过本章将主要介绍 CCG 流。



如果不想实施和维护应用的身份验证部分，并信任第三方来管理此过程，那么 Auth0(https://auth0.com/)就是一个出色的商业解决方案，对于基于微服务的应用，它提供了需要的所有 API。

实现身份验证微服务前，首先需要了解基于令牌的身份验证的原理。如果能正确理解下一节，那么掌握 OAuth2 中的其他内容将更容易。

7.2 基于令牌的身份验证

前面提到，在没有任何用户参与的情况下，当服务想要访问另一个服务时，可使用 CCG 流。

CCG 的初衷是服务可像用户那样得到身份验证服务的验证，并获得一个令牌，然后可使用这个令牌与其他服务交互。

令牌类似于密码，它证明访问特定资源是得到许可的。无论是用户还是微服务，只要拥有资源可识别的令牌，就拥有了访问该资源的密钥。

令牌可保存任何信息，这点在身份验证和授权过程中非常有用。其中包括：

- 用户名或 ID(如果与上下文相关)；
- 操作范围，指示调用者能执行什么操作 (读取、写入等)；
- 时间戳，指示令牌何时发布；

- 过期时间戳，指示令牌多长时间内有效。

令牌通常是一个独立凭证，用作请求服务的许可。“独立”意味着服务将能验证令牌而不必调用外部资源，这是避免增加服务间依赖的绝佳方法。基于令牌的实现方式，一个令牌还可用来访问不同的微服务。

OAuth2 为其令牌使用 JWT 标准。



虽然 OAuth2 未强制要求使用 JWT 标准，但 JWT 标准恰如其分地满足了 OAuth2 的需求。

7.2.1 JWT 标准

RFC 7519 描述的 JSON Web Token(JWT)是令牌的通用标准。

这里，令牌是由三部分组成的长字符串，各部分之间用点分隔：

- header：提供有关令牌的信息，如使用哪种哈希算法；
- payload：提供实际数据；
- signature：提供令牌的签名哈希值，用于检查是否合法。

JWT 令牌使用 base64 编码，所以可用在查询字符串中。

这里是一个编码格式的 JWT 令牌：


```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJ1c2VyIjoidGFyZWsifQ
.
OeMWz6ahNsf-TKg8LQNdNMnFHNtReb0x3NMs0eY64WA
```




为方便显示，上述令牌的每部分都使用换行符进行分隔，而原始令牌是单行的。

如果用 Python 进行解码：

```
>>> import base64
>>> def decode(data):
...     # adding extra = for padding if needed
...     pad = len(data) % 4
...     if pad > 0:
...         data += '=' * (4 - pad)
...     return base64.urlsafe_b64decode(data)
```

```
...
>>> decode('eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9')
b'{"alg":"HS256","typ":"JWT"}'
>>> decode('eyJ1c2VyIjoidGFyZWsifQ')
b'{"user":"tarek"}'
>>> decode('OeMWz6ahNsf-TKg8LQNdNMnFHNtReb0x3NMs0eY64WA')
b'9\xe3\x16\xcf\xa6\xa16\xc7\xfeL\xa8<-
\x03]4\xc9\xc5\x1c\xdbQy\xbd1\xdc\xd3,\xd1\xe6:\xe1'
```

除了 signature 部分外，JWT 令牌的每一块都是 JSON 映射。header 通常包含 typ 键和 alg 键。typ 键表示这是 JWT 令牌，alg 键则表示使用哪种哈希算法。

下面的 header 例子使用的哈希算法参数是 HS256，它代表 HMAC-SHA256 算法：

```
{"typ": "JWT", "alg": "HS256"}
```

payload 包含需要的全部内容，每个字段在 RFC 7519 术语中称为 JWT 声明。

RFC 定义了令牌可能包含的预定义声明列表，称为注册声明名称(Registered Claim Names)。以下是其中一部分：

- iss：是令牌的发布者，是生成令牌的实体名称。通常是一个完全限定的主机名，因此客户端可用这个属性请求/.well-known/jwks.json 来获得公钥。
- exp：是到期时间，用来告知令牌何时失效。
- nbt：是不早于时间，用来告知在何时之前令牌是无效的。
- aud：是受众，用来告知谁是令牌的收件人。
- iat：是令牌发布时间，用来告知令牌何时发布。

下面的 payload 例子提供自定义的 user_id 值，以及可让令牌在发布后 24 小时有效的时间戳。一旦生效，该令牌能使用 24 小时：

```
{
    "iss": "https://tokendealer.example.com",
    "aud": "runnerly.io",
    "iat": 1488796717,
    "nbt": 1488883117,
    "exp": 1488969517,
    "user_id": 1234
}
```

在控制令牌的有效时长方面，这些 header 配置提供了极大的灵活性。

根据微服务的特性，令牌的生存时间(Time-To-Live)可能极短，也可能长到无限。例如，对于微服务之间的交互，在系统中为避免总是重新生成令牌，最好使有效时间

长一些。另一种情况，如果令牌提供给外部请求者，那么有效时间短一些更安全。

JWT 令牌的最后一部分是 signature，它包含 header 和 payload 的签名哈希值。用于签名和哈希的算法有很多。有些基于密钥，有些基于公钥和私钥对。

下面介绍如何在 Python 中使用 JWT 令牌。

7.2.2 PyJWT

在 Python 中，要生成和读回 JWT 令牌，PyJWT(https://pyjwt.readthedocs.io/)库提供了需要的所有工具。

一旦使用 pip 安装了 PyJWT (以及 cryptography)，就能用 encode()方法和 decode()方法来创建令牌了。

下例使用 HMAC-SHA256 算法创建一个 JWT 令牌，然后将其读回。在读取令牌时会根据提供的密钥验证 signature：

```
>>> import jwt

>>> def create_token(alg='HS256', secret='secret', **data):
...     return jwt.encode(data, secret, algorithm=alg)
...
>>> def read_token(token, secret='secret', algs=['HS256']):
...     return jwt.decode(token, secret)
...
>>> token = create_token(some='data', inthe='token')
>>> print(token)
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpbnRoZSI6InRva2VuIiwic29t
ZSI6ImRh
dGEifQ.oKmFaNV-C2wHb_WaMAfIGDqBPnOCyOzVf-JWvh-6bRQ'
>>> read = read_token(token)
>>> print(read)
    {'inthe': 'token', 'some': 'data'}
```

提示： create_token()方法使用算法参数调用 jwt.decode()，算法参数可确保使用正确算法来验证令牌。这是一个很好的实践，可防止恶意令牌诱骗服务器使用意外的算法；可访问 https://auth0.com/blog/critical-vulnerabilities-in-json-web-token-libraries/了解详情。

执行此代码时，令牌将以压缩和未压缩格式显示。

如果使用注册声明中的一项，PyJWT 将根据声明要求进行控制。例如，如果提供

exp 字段但令牌已过期，PyJWT 将抛出错误。

当只有几个微服务运行时，使用密钥来进行签名和验证签名是可行的；但如果微服务数量较多，它很快就会出现问题；所有服务都需要验证签名，因此需要共享密钥。这时若改动密钥，如何安全地对这么多微服务进行修改将成为一项挑战。

而且基于共享的密钥进行身份验证还有一个弱点。如果攻击者入侵某个服务并盗走密钥，整个身份验证系统将遭受损害。

更好的解决方案是使用由公钥和私钥组成的非对称密钥。令牌发布者使用私钥来签署令牌，然后任何人都可利用公钥来确认签名来自发布者。

当然，如果攻击者可获取私钥，或刻意欺骗验证令牌的服务，那么系统依然会被攻破。

但大多数情况下，可使用公钥/私钥对降低在身份验证过程受到攻击的可能性。由于身份验证微服务是系统中唯一拥有私钥的地方，也便于给身份验证微服务添加更多安全措施，如将这种敏感的服务部署在严格控制访问的防火墙环境中。

下面实践一下如何创建非对称密钥。

7.2.3　基于证书的 X.509 身份验证

X.509 标准(https://en.wikipedia.org/wiki/X.509)用于保护 Web 应用。每个使用 SSL 的站点(基于 HTTPS 来提供服务)都在其 Web 服务器上有一个 X.509 证书，并用证书来实时地加密和解密数据。

这些证书由 CA(Certificate Authority)颁发，当浏览器打开一个显示证书的页面时，必须从浏览器支持的 CA 获取证书。

CA 只允许数量有限的受信任实体生成和管理证书，降低了危害证书的风险；CA 应该独立于使用证书的公司。

因为任何人都可从 shell 创建自签名证书，所以如果不确定是否该信任一个证书，可很容易地终止请求。如果证书由一个浏览器信任的 CA 颁发，例如 Let's Encrypt(https://letsencrypt.org/)，那么证书是合法的。



提示：对于我们的微服务，如果拥有本节介绍的架构中的每一部分，使用自签名的证书就可以了。不过，如果微服务需要向第三方公开，或需要使用第三方的服务，最好依赖一个受信任的 CA。Let's Encrypt 非常好用而且免费。此项目旨在确保 Web 的安全，通过扩展，只要有自己的域名就可用它来保护微服务。

让我们尝试创建一个自签名的证书，然后分析如何用它给 JWT 令牌签名。

在 shell 中，可使用 openssl 命令来创建证书，并从证书中提取公钥和私钥对。



提示：如果使用最新的macOS操作系统，需要从brew安装openssl，因为openssl被macOS移除了。

```
$ openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days
365
Generating a 4096 bit RSA private key
..........................++
..........................++
writing new private key to 'key.pem'
Enter PEM pass phrase:
Verifying - Enter PEM pass phrase:
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields, but you can leave some blank
  For some fields, there will be a default value,

  If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:FR
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (e.g., company) [Internet Widgits Pty Ltd]:Runnerly

  Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:Tarek
Email Address []:tarek@ziade.org

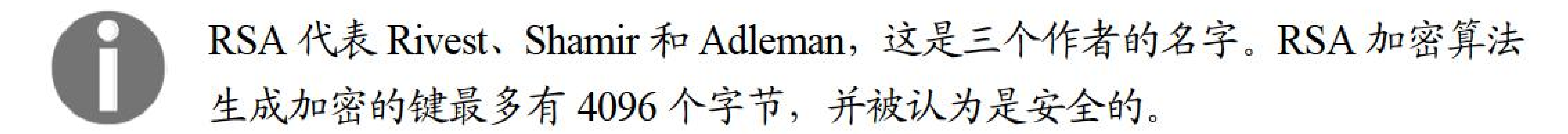
$ openssl x509 -pubkey -noout -in cert.pem > pubkey.pem

$ openssl rsa -in key.pem -out privkey.pem
Enter pass phrase for key.pem:
writing RSA key
```

这三个调用生成 4 文件：

- cert.pem 文件包含证书；
- pubkey.pem 文件包含从证书提取的公钥；

- key.pem 文件包含 RSA 私钥，加密的；
- privkey.pem 文件包含 RSA 私钥，非加密的。



RSA 代表 Rivest、Shamir 和 Adleman，这是三个作者的名字。RSA 加密算法生成加密的键最多有 4096 个字节，并被认为是安全的。

到此，我们可使用 pubkey.pem 和 privkey.pem 在 PyJWT 脚本中进行签名和验证令牌的签名，验证过程使用 RSASSA-PKCS1-v1_5 签名算法和 SHA-512 哈希算法：

```
import jwt

with open('pubkey.pem') as f:
    PUBKEY = f.read()

with open('privkey.pem') as f:
    PRIVKEY = f.read()

def create_token(**data):
    return jwt.encode(data, PRIVKEY, algorithm='RS512')

def read_token(token):
    return jwt.decode(token, PUBKEY)

token = create_token(some='data', inthe='token')
print(token)

read = read_token(token)
print(read)
```

结果与此前类似，但得到一个更大的令牌：


```
b'eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzUxMiJ9.eyJzb21lIjoiZGF0YSIsImludGhl
IjoidG9rZW4ifQ.VHKP2yO1dCUrS5YAOCZsGXF_mesMJNNYcnBHe4mFiPpBDCbMhrI8
h10vr1BaCiN8rVEMcUXQ4Gc7183w6ga3spyEzONg3-Sv-eId4rPbTqbbmPErrnWPRIH
9hQMHsMebVOlI9lOvNmVJ3DIEmV4riqRluJMIFYuy_A7fB2r8IqeHBfrsEPWmvw2_tI
Z3V3dJGU4ZBkn8zdzgfbou_LHc28_dyC32kR2Ec1nsRV3zRffEjx60cjzmNNFqB9kYZ
Hun0IIzBqdh0IiRxPF4rgYG3oBKJXP3u2uyfBifNy3Bz4bMPJ8iRRmQleciyFdzDkm7
J4SAyz5I0TKHSPOZA-9x6dgacQ9w_JAtmElH7u8_ES_2TxmvbBLqsXIzghAhG10CL79
UeSKeXMTjc8DOQrIbWmaRCIbPy9AdlIJQxqul4UnCoUhUQ6PZwD6CEuaZTjKdPvql7n
_-u1Tjrw7e339WC9QZS5DFCzMe2F0TY-kI52-AaNEoRaO8oSCwW3E7u-NcStbD019Md
```

```
X3bxN0FdNvL62BUDqqxind7TFF7YFX3zTxTu15Pex2F64YvnhG1CDk337htROt8B9vH
8CIUWo_2ujkair8zCdd9sfIdssOGFDnawIX2NPGd4vZ1dpw0DwHBaXw0gP8zzcRAsuZ
7rfNMZeJTH6gB-kMc5UKf26nAc'
{'some': 'data', 'inthe': 'token'}
```

注意，需要为每个请求添加超过 700 字节的数据，如果要减少网络开销，记住“基于密钥的 JWT 令牌技术”是一个选项。

我们已经学习了如何生成 JWT 令牌，现在开始实现身份验证微服务，即 TokenDealer。

7.2.4 TokenDealer 微服务

构建身份验证微服务的第一步是实现 CCG 流所需的一切。对于 CCG 流，应用从需要令牌的服务接受请求，然后根据需要生成令牌。令牌的有效期是 1 天。

这个服务将是唯一可用私钥签名令牌的服务，并对其他要验证令牌的服务提供公钥。这个服务也是唯一可保存所有客户 ID 和密钥的地方。

一旦服务获取到令牌，就可极大地简化实现，还可访问系统生态中的任何服务。当通过令牌访问一个服务时，可在本地或调用 TokenDealer 进行验证。如果选择第一种方案，将减少一个网络往返，但使用 JWT 令牌时会增加 CPU 开销，这种方案在某些场景存在隐患。例如，如果微服务正在执行 CPU 密集型工作，再添加检查令牌的工作就需要选用有较大 CPU 的服务器，这会增加一些成本。

好在，有两种选项可供选择。

为实现我们描述的一切，这个微服务需要创建 3 个 API：

- GET/.well-known/jwks.json：这个 API 提供公钥，使用 JSON Web Key 格式发布，这种格式由 RFC 7517(https://tools.ietf.org/html/rfc7517)描述，当其他微服务想要验证令牌时，会调用这个 API。
- POST/oauth/token：这个 API 通过给定的凭证返回令牌。OAuth RFC 中使用了/oauth 前缀，此后，添加/oauth 前缀成为一个被广泛采用的标准。
- POST/verify_token: 这个 API 返回令牌有效性，如果给定令牌非法，则返回 400 错误。

使用 https://github.com/Runnerly/microservice 上的微服务源代码骨架，可创建一个非常简单的 Flask blueprint，其中实现了这三个 API。

下面介绍最重要的一个 API，即 POST/oauth/token。

实现 POST/oauth/token

对于 CCG 流，为使令牌的服务发送 POST 请求，body 中会包含以下字段的 URL 编码:

- client_id：一个字符串，用来唯一标识请求者。
- client_secret：用来验证请求者身份的密钥。它应该是一个随机字符串，由前端生成并注册在 auth 服务上。
- grant_type：授权类型，必须是 client_credentials。

为简化实现，我们设定了一些假设：

- 在 Python 映射中保存了密钥列表。
- client_id 是微服务名。
- 密钥由 binascii.hexlify(os.urandom(16))生成。

身份验证部分只确保密钥是合法的，此后服务会创建一个令牌并返回：

```
import time
from flask import request, current_app, abort, jsonify
from werkzeug.exceptions import HTTPException
from flakon import JsonBlueprint
from flakon.util import error_handling
import jwt

home = JsonBlueprint('home', __name__)

def _400(desc):
    exc = HTTPException()
    exc.code = 400
    exc.description = desc
    return error_handling(exc)

_SECRETS = {'strava': 'f0fdeb1f1584fd5431c4250b2e859457'}

def is_authorized_app(client_id, client_secret):
    return compare_digest(_SECRETS.get(client_id), client_secret)

@home.route('/oauth/token', methods=['POST'])
def create_token():
    key = current_app.config['priv_key']
    try:
```

```
        data = request.form
        if data.get('grant_type') != 'client_credentials':
            return _400('Wrong grant_type')

        client_id = data.get('client_id')
        client_secret = data.get('client_secret')
        aud = data.get('audience', '')

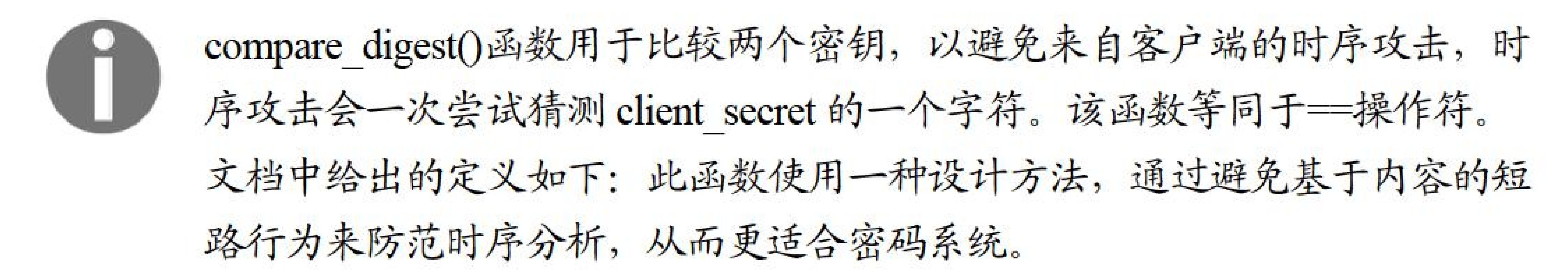
        if not is_authorized_app(client_id, client_secret):
            return abort(401)

        now = int(time.time())

        token = { 'iss': 'https://tokendealer.example.com',
                  'aud': aud,
                  'iat': now,
                  'exp': now + 3600 * 24}

        token = jwt.encode(token, key, algorithm='RS512')
        return {'access_token': token.decode('utf8')}
    except Exception as e:
        return _400(str(e))
```

create_token()视图使用私钥，私钥放在应用配置的 priv_key 下。



compare_digest()函数用于比较两个密钥，以避免来自客户端的时序攻击，时序攻击会一次尝试猜测 client_secret 的一个字符。该函数等同于==操作符。文档中给出的定义如下：此函数使用一种设计方法，通过避免基于内容的短路行为来防范时序分析，从而更适合密码系统。

上面代码里的 blueprint 就是我们需要的全部，其中一对 key 用来运行微服务，这个微服务负责生成独立的 JWT 令牌，令牌供所有需要身份验证的微服务使用。



可在 https://github.com/Runnerly/tokendealer 中找到 TokenDealer 微服务的完整源代码，在其中能找到其他两个视图是如何实现的。

微服务可提供与生成令牌相关的更多功能。例如，管理范围的能力，确保微服务 A 不允许生成一个供 B 使用的令牌；或管理一个服务白名单，授权这些服务请求某些令牌。

对于基于令牌的身份验证系统，我们实现的模式在微服务环境中仅是基础，当然你也可自行开发实现模式，不过对当前 Runnerly 应用来说，已经足够好了。

在图 7-2 中，训练计划、数据服务和竞赛都可使用 JWT 令牌来限制对终端的访问：

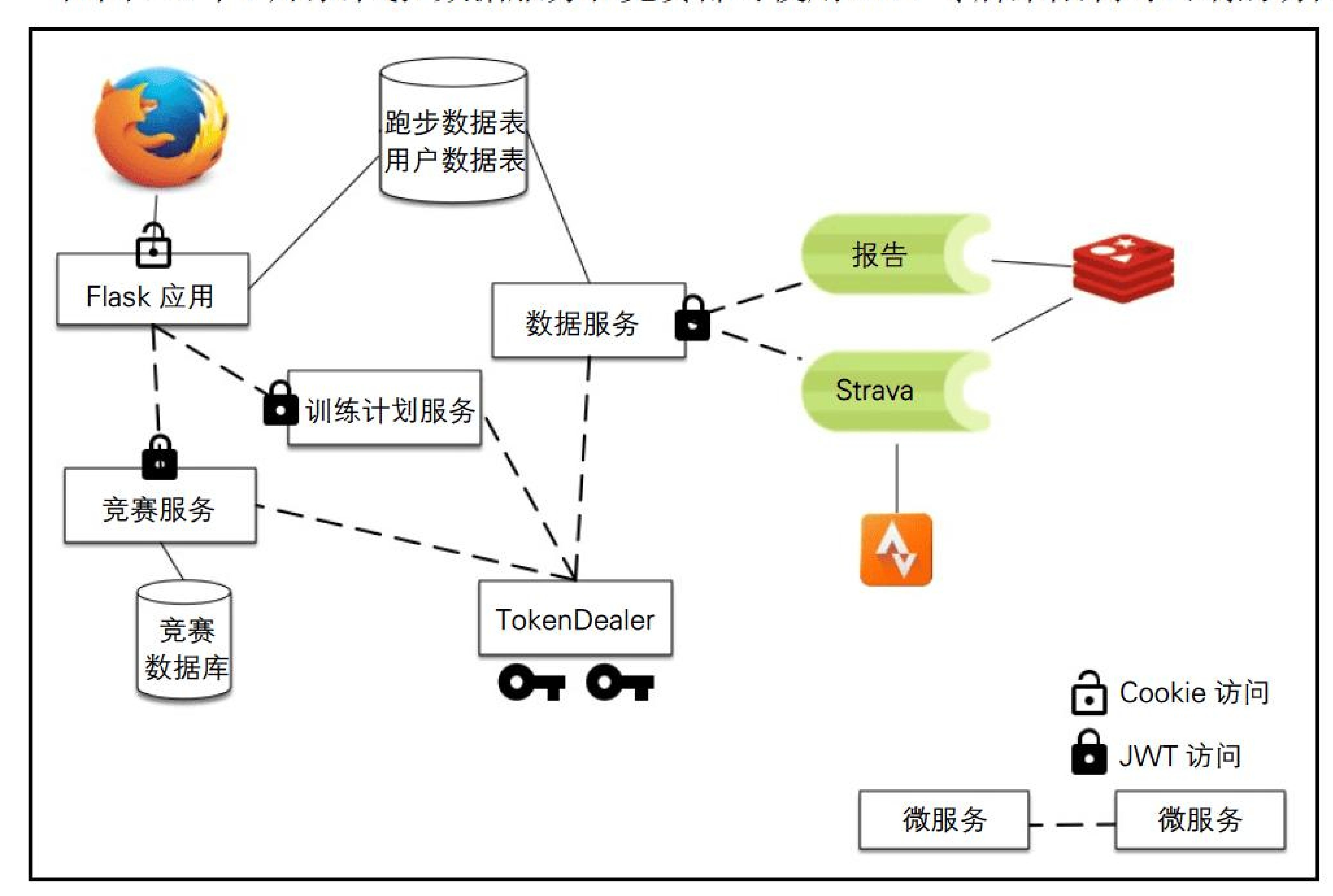


图 7-2　使用 JWT 令牌来限制对终端的访问

图中的JWT访问表示：服务需要一个JWT 令牌。这些服务可通过调用TokenDealer来验证令牌。图中的Flask应用需要以用户身份从TokenDealer获取令牌(图中没有显示这个链接)。

现在已经实现了 CCG 的 TokenDealer 服务，下一节介绍服务如何使用它。

7.2.5　使用 TokenDealer

在 Runnerly 中，数据服务到 Strava 职程的连接③是一个需要进行身份验证的好例子。通过严格限制授权的数据服务来添加跑步活动，如图 7-3 所示。

要给这个链接添加身份验证，可通过下列 4 个步骤完成：

(1) TokenDealer 为 Strava 职程保存 client_id 和 client_secret 对，然后分享给 Strava 职程的开发者①。

(2) Strava 职程使用 client_id 和 client_secret 向 TokenDealer 请求令牌②。

(3) Strava 职程在每次向 Data Service 发送请求时都带令牌③。

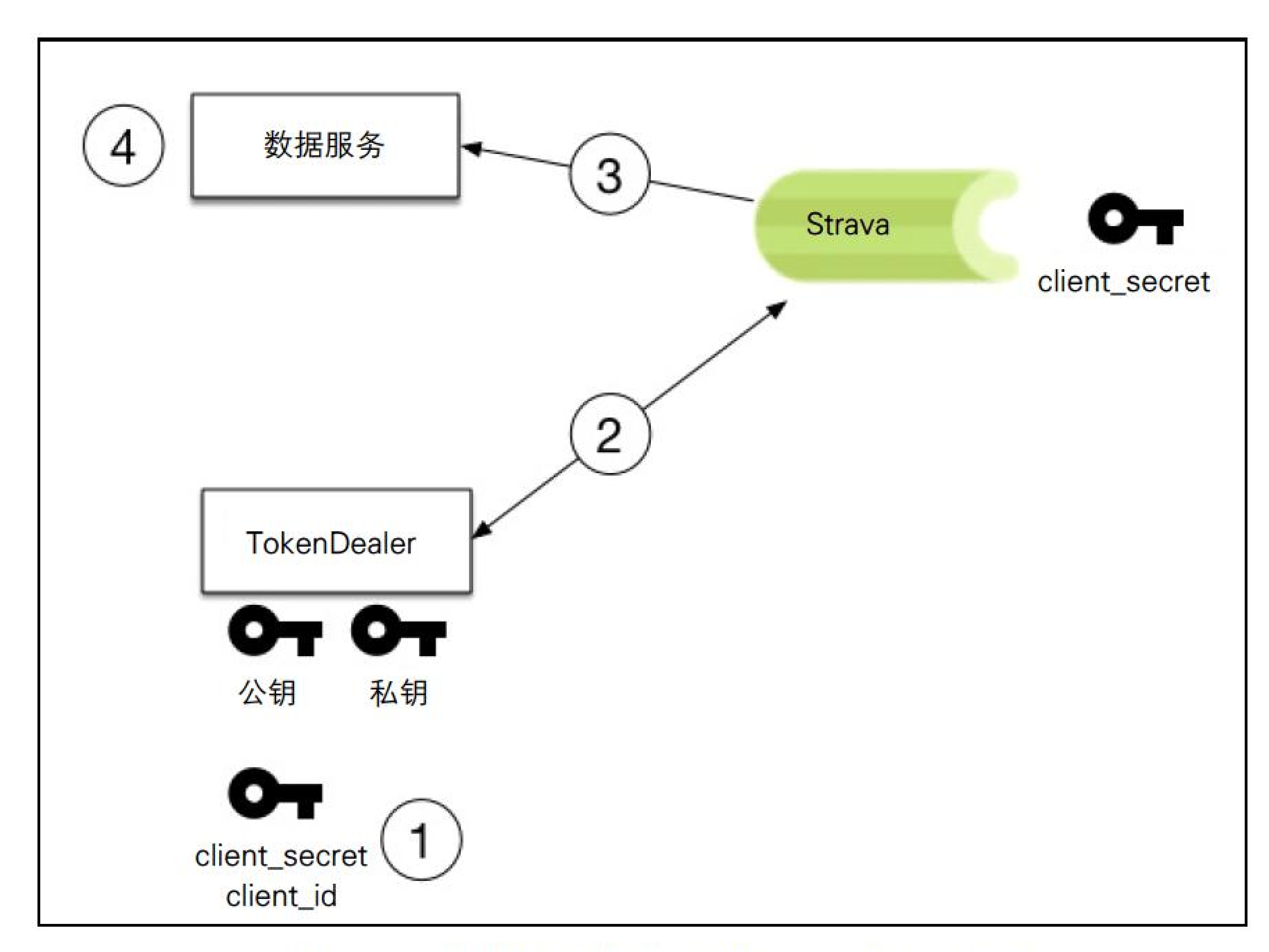


图 7-3　通过严格限制授权的数据服务来添加跑步活动

(4) 数据服务通过调用 TokenDealer 来验证令牌，或执行一个本地 JWT 验证。

在完整实现中，第一步是半自动的。通常在身份验证服务的管理页面生成客户密钥。这个密钥供 Strava 微服务的开发者使用。

之后，服务可每次在需要时获取一个新令牌(因为是第一次获取，或之前获取的令牌过期了)，在请求数据服务时，将令牌放在身份验证头信息中。

下面列举这样一个例子，使用了 requests 库(示例中有一个运行在 localhost:5000 上的 TokenDealer，还有一个运行在 localhost: 5001 上的数据服务)：

```
import requests

server = 'http://localhost:5000'
secret = 'f0fdeb1f1584fd5431c4250b2e859457'

data = [('client_id', 'strava'),
        ('client_secret', secret),
        ('audience', 'runnerly.io'),
        ('grant_type', 'client_credentials')]

def get_token():
    headers = {'Content-Type': 'application/x-www-form-urlencoded'}
    url = server + '/oauth/token'
```

```
    resp = requests.post(url, data=data, headers=headers)
    return resp.json()['access_token']
```



注意，/oauth/token 接受的是加密数据而非 JSON，这是一个标准化实现。

当代码调用 Data Service 时，get_token()函数获取一个令牌，令牌用在 Authorization 头信息中。

```
_TOKEN = None

def get_auth_header(new=False):
    global _TOKEN
    if _TOKEN is None or new:
        _TOKEN = get_token()
    return 'Bearer ' + _TOKEN

_dataservice = 'http://localhost:5001'
def _call_service(endpoint, token):
    # not using session etc, to simplify the reading :)
    return requests.get(_dataservice + '/' + endpoint,
                        headers={'Authorization': token})
def call_data_service(endpoint):
    token = get_auth_header()
    resp = _call_service(endpoint, token)
    if resp.status_code == 401:
        # the token might be revoked, let's try with a fresh one
        token = get_auth_header(new=True)
        resp = _call_service(endpoint, token)
    return resp
```

如果调用 Data Service 导致 401 响应，call_data_service()函数将尝试获取一个新令牌。

“当 401 时刷新令牌”模式可在所有微服务中用来自动生成令牌。

这里主要介绍服务之间的身份验证。可在 GitHub 代码库的 Runnerly 项目中找到基于 JWT 身份验证的完整实现，并将其作为身份验证流程的基础。

下一节将介绍保护 Web 服务的另一个重要方法：Web 应用防火墙。

7.3 Web 应用防火墙

将 HTTP 端点向其他人公开时，希望调用者能按期望行事。每个 HTTP 会话都应遵循服务中设定的方案。

但在现实世界中，情况并非总是如此。如果调用者有缺陷或只是没有正确地调用服务，期望的行为应该是返回一个 4*xx* 响应，并向客户端解释请求被拒绝的原因。对于攻击者发送的恶意请求也应该如此对待。任何意外行为都应该被驳回。

OWASP(Open Web Application Security Project，开源 Web 应用安全项目，https://www.owasp.org)是一个优秀资源，有助于了解如何保护 Web 应用免受不良行为的侵害。甚至为 ModSecurity 工具包中的 Web 应用框架提供一组规则，以避免大部分攻击。

在基于微服务的应用中，任何发布到网上的东西都可能受到攻击。但不同于单体应用，大部分系统并非直接通过 HTML 用户交互界面或公开的 API 与用户打交道，这就缩小了潜在攻击的范围。

本节将介绍如何给基于 JSON 的微服务提供基本保护。

在此之前，先介绍一些最常见的攻击：

- **SQL 注入**：攻击者在请求中发送原始 SQL 语句。如果服务器使用某些请求内容(通常是参数)来生成 SQL 查询，可能在数据库上执行攻击者的请求。在 Python 中，如果使用 SQLAlchemy 并避免使用原始语句，将是安全的。如果使用原始 SQL，要确保每个变量都正确地加了引号。稍后将继续讨论这个话题。
- **跨站点脚本(XSS)**：这个攻击只发生在显示 HTML 的 Web 页面上。攻击者使用一些查询属性尝试在页面上注入 HTML 片段，以诱使用户认为是在合法网站上执行操作。
- **跨站点请求伪造(XSRF/CSRF)**：此攻击重用用户在其他网站的凭证来攻击服务。典型的 CSRF 攻击发生在发起 POST 请求时，例如，一个恶意站点给用户显示一个链接，欺骗用户在站点上使用已存在的凭证发起 POST 请求。

其他很多攻击都瞄准 PHP 系统，因为很容易就能找到很多 PHP 应用，当调用服务器时，这些应用可使用无效的用户输入。诸如本地文件包含(Local File Inclusion，LFI)、远程文件包含(Remote File Inclusion，RFI)或远程代码执行(Remote Code Execution，RCE)是常见的攻击方式；利用客户端输入或外泄的服务器文件，这些攻击会欺骗服务器执行一些操作。当然，这些攻击也发生在 Python 应用中，但众所周知的 Python 框架有内置的保护机制来避免这些攻击。

无论是不是恶意客户端，坏请求并非始终采用滥用系统的方式。它可通过发送合法请求来损害系统，比如由于所有资源都用来处理攻击者的请求而导致拒绝服务(Denial of Service，DoS)。当客户端有重发功能并自动重新请求同一 API 时，分布式系统中有时会发生这个问题。如果客户端没有对请求进行节流，这些合法的客户端最终可能导致服务过载。

通常很难在服务端添加保护来防范此类客户端，要保护好整个微服务堆，还有很长的路要走。

本节重点介绍如何创建一个基本 WAF，它将明确拒绝对服务提出太多请求的客户端。



本节并非创建一个完整的 WAF，而是引导你很好地理解如何实现和使用 WAF。不过，对基于 JSON 的微服务来说，使用诸如 ModSecurity 的功能完备的 WAF 有些小题大作了。

可在 Flask 微服务中构建自己的 WAF，但如果所有流量都经过它，会增加很多开销。一个更好的解决方案是直接依靠 Web 服务器。

OpenResty-Lua 和 nginx

OpenResty(http://openresty.org/en/)是一个内嵌了 Lua(http://www.lua.org/)解释器的 nginx 分发器，Lua 可用来编写 Web 服务器脚本。

Lua 是支持动态类型的卓越编程语言，它有一个速度极快的轻量级解释器。该语言提供一整套功能，并具有内置的异步功能。可使用普通 Lua 脚本编写协同程序。

在 Python 开发者眼中，Lua 非常 Python 化，一旦熟悉基本语法，几小时内就可开始编写脚本。它有函数、类以及一个你很熟悉的标准库。

如果安装了Lua(http://www.lua.org/start.html)，就可使用Lua Read Eval Print Loop(REPL)来编程，它与Python的操作方式完全一样：

```
$ lua
Lua 5.1.5 Copyright (C) 1994-2012 Lua.org, PUC-Rio
> io.write("Hello world\n")
Hello world
> mytable = {}
> mytable["user"] = "tarek"
> = mytable["user"]
tarek
> = string.upper(mytable["user"])
TAREK
```



要探索 Lua 语言，可访问 http://www.lua.org/docs.html 了解更多相关信息。

Lua 通常作为一种语言选项，被嵌入编译的应用中。它对占用的内存很少，并且允许添加速度很快的动态脚本。这就是在 OpenResty 中使用 Lua 所发生的事情。不同于在构建 nginx 模块时需要将脚本和 nginx 一起编译，你可使用 Lua 脚本扩展 Web 服务器，然后直接和 OpenResty 一起部署。

从 nginx 配置中调用某些 Lua 代码时，OpenResty 使用的 LuaJIT(http://luajit.org/)解释器将高效地运行它们，并不比 nginx 代码慢。一些性能基准对比实验发现，某些情况下 Lua 可能比 C 或 C++更快(请参阅 http://luajit.org/performance.html)。

在 nginx 中添加的函数协同程序将在 nginx 中异步运行，因此当服务器收到大量并发请求时，服务器的开销也很小，这正是我们想要的 WAF。

OpenResty 作为 Docker 镜像和一些 Linux 版本的包，也可在本地编译，相关内容请参考 http://openresty.org/en/installation.html。在 macOS 上，可使用 Brew 和 brew install openresty 命令。

一旦 OpenResty 安装完成，将获得 openresty 命令，可像 nginx 一样用它为应用提供服务。

下例中，nginx 配置将代理对 Flask 应用的调用，该应用运行在 5000 端口上：

```
daemon off;
worker_processes 1;
pid openresty.pid;
error_log /dev/stdout info;
events {
  worker_connections 1024;
}
http {
  include         mime.types;
  default_type    application/octet-stream;
  sendfile          on;
  keepalive_timeout  65;
  access_log /dev/stdout;
  server {
    listen          8888;
    server_name localhost;
    location / {
      proxy_pass http://localhost:5000;
```

```
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
      }
    }
  }
```

此配置可与 openresty 命令行一起使用，将在端口 8888 上运行一个前台进程(守护进程)，它会代理所有请求，然后传给运行在 5000 端口的 Flask 应用。

```
$ openresty -c resty.conf
2017/07/14 12:10:12 [notice] 49704#524185: using the "kqueue" event
method
2017/07/14 12:10:12 [notice] 49704#524185: openresty/1.11.2.3
2017/07/14 12:10:12 [notice] 49704#524185: built by clang 8.0.0
(clang-800.0.38)
2017/07/14 12:10:12 [notice] 49704#524185: OS: Darwin 16.6.0
2017/07/14 12:10:12 [notice] 49704#524185: hw.ncpu: 4
2017/07/14 12:10:12 [notice] 49704#524185: net.inet.tcp.sendspace:
1042560
2017/07/14 12:10:12 [notice] 49704#524185: kern.ipc.somaxconn: 2048
2017/07/14 12:10:12 [notice] 49704#524185: getrlimit(RLIMIT_NOFILE):
7168:9223372036854775807
2017/07/14 12:10:12 [notice] 49704#524185: start worker processes
2017/07/14 12:10:12 [notice] 49704#524185: start worker process 49705
```

注意，此配置也可在普通 nginx 服务器中使用，因为还没有使用任何 Lua 脚本。这就是 OpenResty 的好处：它是一个插入式 nginx 替代品，可运行现有配置文件。



本节的代码和配置可在 https://github.com/Runnerly/waf 上找到。

当请求到达时，可在不同时刻调用 Lua，两个相关项如下：

- **access_by_lua_block**：在生成响应前，对每个传入的请求都调用此项。这是可在 WAF 中构建访问规则的地方。
- **content_by_lua_block**：使用 Lua 生成响应。

下一节将介绍如何对传入请求进行速率限制。

1. 速率和并发限制

速率限制指统计服务器在一段时间内接受的请求数量，并在达到限制时拒绝新请求。

并发限制指统计 Web 服务器向同一远程用户提供的并发请求数量，并在达到定义的阈值时开始拒绝新请求。由于许多请求可同时到达服务器，并发限制器的阈值应当留有较小的余量。

两个限制都用相同的技术实现。下面介绍如何构建一个并发限制器。

OpenResty 有一个使用 Lua 编写的速率限制库 lua-resty-limit-traffic(https://github.com/openresty/lua-resty-limit-traffic)，可在 acces_by_lua_block 部分使用它。

这个功能使用 Lua Shared Dict，这是同一个进程内所有 nginx 工作器都共享的内存映射。使用内存字典意味着速率限制将在进程级别工作。



提示：因为每个服务节点通常部署一个 nginx，所以速率限制将发生在每个 Web 服务器上。因此，如果给同一微服务部署多个节点来实现负载均衡，就必须在设置阈值时考虑这一点。

下例添加了 lua_shared_dict 定义和 access_by_lua_block 部分来激活速率限制。注意，本示例是项目文档中示例的简化版本：

```
...
http {
  ...
  lua_shared_dict my_limit_req_store 100m;

  server {
    access_by_lua_block {
      local limit_req = require "resty.limit.req"
      local lim, err = limit_req.new("my_limit_req_store",200, 100)
      local key = ngx.var.binary_remote_addr
      local delay, err = lim:incoming(key, true)
      if not delay then
        if err == "rejected" then
          return ngx.exit(503)
          end
        end
      if delay >= 0.001 then
        ngx.sleep(delay)
```

```
        end
      }
      proxy_pass ...
    }
  }
```

access_by_lua_block 部分可当成 Lua 函数，其中可使用 OpenResty 公开的一些变量和函数。例如，ngx.var 是存放所有 nginx 变量的表，ngx.exit()是可直接给客户端返回响应的函数。在本例中，由于速率限制而拒绝一个调用时会返回 503 响应。

每次当请求到达服务器时，库都将 my_limit_req_store 字典传给 resty.limit.req 函数，然后用包含客户端地址的 binary_remote_addr 值调用 incoming()函数。

incoming()函数将使用共享字典为每个远程地址维护激活连接个数，并在该数字达到阈值时送回一个拒绝值，例如，当超过 300 个并发请求时拒绝请求。

如果连接被接受，incoming()函数会送回一个延迟值。Lua 会使用这个延迟值和异步的 ngx.sleep()函数挂起请求。远程客户端线程数未达到阈值 200 时，延迟为 0，远程客户端线程数在 200~300 范围时会出现短暂延迟，因此服务器有机会取消所有在栈内等待的请求。

这种优雅的设计非常高效，可避免服务被许多请求淹没。设置上限也是一个好方法，可避免服务器到达你知道的崩溃点。

例如，如果一些基准确定服务在开始崩溃前无法同时支持超过 100 个请求，则可设置速率限制；让 nginx 拒绝请求，总比让 Flask 微服务继续堆积错误日志和仅为处理拒绝执行无意义的 CPU 计算要好。



提示：本例中计算速率的关键是请求的远程地址头。如果 nginx 服务器本身位于代理后面，则使用的头信息务必包含实际远程地址。否则，将限制单个远程客户端(代理服务器)的速率。这里，远程地址通常在 X-Forwarded-For 的头信息中。

lua-resty-waf(https://github.com/p0pr0ck5/lua-resty-waf)项目能完成与 lua-resty-limit-traffic 同样多的功能，但提供了更多保护。还能读取 ModSecurity 规则文件，因此，在不必使用 ModSecurity 本身的情况下，可使用 OWASP 项目中的规则文件。

2. 其他 OpenResty 功能

OpenResty 附带许多 Lua 脚本，可帮助增强 nginx。有些开发人员甚至使用脚本来直接访问数据。

如果阅读 http://openresty.org/en/components.html 的组件说明页面，可找到一些有用工具，这些工具使 nginx 与数据库、缓存服务器等进行交互。还有一个网站专门向社区发布 OpenResty 组件，网址是 https://opm.openresty.org/。

如果在 Flask 微服务的前面使用 OpenResty，可能还有其他使用场景，可让你将本来在 Flask 应用中的代码转成 OpenResty 中的 Lua 代码。这样做的目的不是将应用逻辑放到 OpenResty 中，而是利用 Web 服务器在调用 Flask 应用前后完成一些事情。

例如，如果使用 Redis 或 Memcache 服务器来缓存 GET 资源，可直接用 Lua 调用它们，添加或重取一个端点的缓存版本。srcache-nginx-module(https://github.com/openresty/srcache-nginx-module)就是一个实现了上述功能的工具，由于使用了缓存，所以减少了直接发给 Flask 应用的 GET 调用。

总结一下本节关于 Web 应用防火墙讨论的内容，OpenResty 是一个强大的 nginx 分发，可用于创建简单的 WAF 来保护微服务，它还提供一些防火墙之上的能力。事实上，如果你才开始用 OpenResty 构建微服务，要感谢 Lua，它为你开启全新的世界。

下一节将重点讨论可在代码级别上执行哪些操作来保护微服务。

7.4 保护代码

上一节介绍了如何设置一个简单 WAF。虽然添加的速率限制功能十分有用，但只能避免一种可能的攻击。一旦你向世界公开应用，就可能遭受各种攻击，你的代码需要抵御这些威胁。

安全代码背后的思想很简单，但在实践中却难做好。有以下两项基本原则：

- 在外部的每个请求操作应用和数据前，都应该审慎地评估它们。
- 应用在系统上执行的所有操作都应该是定义明确且范围有限的。

下面看看如何遵循这两个原则进行编码实践。

7.4.1 断言传入的数据

第一个原则是断言传入的数据，这意味着在不确定会产生什么影响时，不应该盲目执行传入请求。

例如，如果有一个 API 将让调用者删除数据库的一行，就需要明确是否允许这个调用者这样做。这就是前面添加身份验证和授权的原因。

还有其他方法可侵入。例如，如果有一个 Flask 应用从传入的请求中抓取 JSON 数据，然后将数据推送到数据库，就应该校验传入的请求中是否含有期望的数据，而

不是将数据盲目地传给数据库后台。因此，使用 Swagger 定义数据接口并用这个定义校验传入的数据是很有意义的。

微服务通常使用 JSON 格式，但如果使用模板方式，这就是另一个需要小心之处，要注意模板是如何处理变量的。

当模板盲目执行一些 Python 语句时，一种可能的攻击是服务器端模板注入 (Server-Side Template Injection)。2016 年，在基于 Jinja2 模板的 Uber 网站上发现了这样一个注入漏洞：漏洞在执行模板之前已完成了原始格式化。

代码类似于下面这个小应用：

```
from flask import Flask, request, render_template_string

app = Flask(__name__)

SECRET = 'oh no!'

_TEMPLATE = """\
Hello %s

Welcome to my API!
"""


class Extra(object):
    def __init__(self, data):
        self.data = data

@app.route('/')
def my_microservice():
    user_id = request.args.get('user_id', 'Anynomous')
    tmpl = _TEMPLATE % user_id
    return render_template_string(tmpl, extra=Extra('something'))
```

由于使用原始%s 在模板上进行预格式化处理，这个视图在应用中产生了一个巨大的安全漏洞，因为它允许攻击者在 Jinja 脚本执行前注入代码。

在下例中，user_id 变量发生了安全攻击，它从模块中读取全局变量 SECRET：

```
http://localhost:5000/?user_id={{extra.__class__.__init__.__globals
__["SECRET"]}}
```

因此，当显示视图时避免手工格式化非常重要。如果要评估模板中不信任的代码，可使用 Jinja 沙盒，请参考 http://jinja.pocoo.org/docs/latest/sandbox/。对于正在评估的对象，此沙盒将拒绝任何对其方法和属性的访问。例如，如果要在模板中传递可调用的对象，要确保诸如__class__的属性是不能使用的。

不过，由于语言的特性，很难正确配置 Python 沙盒。很容易误配置一个沙盒，或者沙盒本身可能被语言的新版本破坏。最安全的方法是避免同时评估不受信任的代码，并使模板不直接依赖于传入的数据。

另一个常发生注入攻击的地方是 SQL 语句。如果某些 SQL 查询是用原始 SQL 语句生成的，则会将应用暴露在 SQL 注入漏洞下。

在下例中，一个使用用户 ID 作为查询参数的简单 select 查询可被注入额外的 SQL 语句，如 insert 语句。这样，攻击者很容易就能侵入数据库服务器：

```
import pymysql

connection = pymysql.connect(host='localhost', db='book')

def get_user(user_id):
    query = 'select * from user where id = %s'
    with connection.cursor() as cursor:
        cursor.execute(query % user_id)
        result = cursor.fetchone()
    return result

extra_query = """\
insert into user(id, firstname, lastname, password)
values (999, 'pnwd', 'yup', 'somehashedpassword')
"""

# this call will get the user, but also add a new user!
get_user("'1'; %s" % extra_query)
```

通过给原始 SQL 查询的参数值加引号，就可防止这个漏洞。在 PyMySQL 中，通过参数来传递值就能避免这个问题：

```
def get_user(user_id):
    query = 'select * from user where id = %s'
    with connection.cursor() as cursor:
        cursor.execute(query, (user_id,))
```

```
        result = cursor.fetchone()
    return result
```

每个数据的工具库都有这个功能。因此，只要在构建原始 SQL 时正确使用这些工具库就足够了。

预防重定向漏洞也使用相同的方式。一种常见错误是，假定调用者会被重定向到一个内部页面，并使用普通的URL 作为重定向的地址，为此创建一个登录视图：

```
@app.route('/login')
def login():
    from_url = request.args.get('from_url', '/')
    # do some authentication
    return redirect(from_url)
```

这个视图可将调用者重定向到任何页面，在登录过程中这将是明显的威胁。调用 redirect()时，使用 url_for()函数是避免空字符串的好方式，url_for()函数将在应用领域创建一个链接。

有时需要重定向到第三方，将不能使用 url_for()和 redirect()函数了，它们可能将客户送到不想去的地方。

一个解决方案是创建一个应用允许重定向的第三方域名列表作为白名单，确保任何通过应用或第三方库完成的重定向都用这个白名单进行检查。

使用 after_request()钩子可完成以上解决方案，当 Flask 发送响应时，会调用这个函数。如果应用尝试返回 302，你可根据给定的域名和端口列表来检查地址是否安全：

```
from flask import make_response
from urllib.parse import urlparse

# domain:port
SAFE_DOMAINS = ['github.com:443', 'ziade.org:443']

@app.after_request
def check_redirect(response):
    if response.status_code != 302:
        return response
    url = urlparse(response.location)
    netloc = url.netloc
    if url.scheme == 'http' and not netloc.endswith(':80'):
        netloc += ':80'
    if url.scheme == 'https' and not netloc.endswith(':443'):
```

```
        netloc += ':443'

    if netloc not in SAFE_DOMAINS:
        # not using abort() here or it'll break the hook
        return make_response('Forbidden', 403)
    return response
```

总之，对于传入的数据，应该始终将它们视为可能对系统发起注入攻击的威胁。

7.4.2 限制应用的范围

即使你在保护应用上已经做得很好，传入数据不会造成应用的不良行为，也还应该确保应用本身无法损害微服务生态。

如果微服务被授权与其他微服务进行交互，应该对这些交互进行身份验证，还要限制到最小可用级别。换言之，如果一个微服务正向其他微服务发起读取调用，它就不能执行任何 POST 调用，并限制成只读。

可在 JWT 令牌上定义角色(如读/写)来限制请求范围，在令牌的 permissions 或 scope 键下添加相关信息。例如，当 POST 请求使用的令牌只有读权限时，目标微服务可拒绝请求。

这就是给一个应用授予访问 GitHub 账户权限或 Android 手机权限时发生的事情。会显示应用要做的详细清单，可同意或拒绝这些访问权限。

如果要控制整个微服务生态，还可在系统级别使用严格的防火墙规则，限制与每个微服务交互的 IP 白名单，但这种设置很大程度上取决于在哪里部署应用。在 AWS(Amazon Web Services)云环境中，不需要配置 Linux 防火墙。只需要在 AWS 控制台设置简单访问规则即可。

第 11 章将介绍在亚马逊云上部署微服务的基本知识。

除了网络访问，只要可能，应用可访问的任何其他资源都应受到限制。以 Linux 的 root 身份运行应用并不是好主意，当发生安全问题时，会给予服务过高的权限。

例如，如果应用正在调用系统，调用被一个注入或其他攻击挟持了，这就等于给攻击者提供了控制整个操作系统的后门。

对系统的 root 访问已成为现代部署的间接威胁，因为大多数应用都在运行虚拟机(Virtual Machines，VM)，但即便是一个被监禁的进程，如果不对其进行限制，也可造成很多破坏。如果一个攻击者控制了 VM，则可能进一步控制整个系统。

遵循以下两个规则可缓解这个问题：

- Web 服务进程应由非 root 用户运行。

- 从 Web 服务执行其他进程时要非常谨慎，尽量避免这样做。

对于第一个规则，诸如 nginx 的 Web 服务器的默认行为是使用 www-data 用户和组来运行进程，这样做会阻止这些进程在系统上执行任何操作。类似的规则应当用在 Flask 进程上。你将在第 9 章看到，最佳实践是在 Linux 系统的用户空间中运行服务栈。

对于第二条规则，任何 Python 在调用 os.system()、subprocess 和 multiprocessing 时，都应进行双重校验来避免在系统上进行非预期调用。对于通过本地系统发送电子邮件或通过 FTP 连接到第三方服务器的高级网络模块也是如此。要解决潜在的安全问题，有一种方法可持续检查代码，就是使用 Bandit linter。

7.4.3　使用 Bandit linter

OpenStack 社区(https://www.openstack.org/)创建了一个非常好用且安全的小 linter，称为 Bandit，用来尝试查找不安全的代码(https://wiki.openstack.org/wiki/Security/Projects/Bandit)。

该工具使用 ast 模块解析代码，与 Flake8 或其他 linter 类似。Bandit 会在代码中扫描已知的安全问题。

一旦使用 pip install bandit 命令安装了它，即可运行 Bandit 命令扫描 Python 代码模块。

下面的示例代码包含三个不安全的函数。第一个加载的 YAML 内容可能实例化任意对象，而其后的一些则容易导致注入攻击：

```
import subprocess
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker
import yaml

def read_file(filename):
    with open(filename) as f:
        data = yaml.load(f.read())

def run_command(cmd):
    return subprocess.check_call(cmd, shell=True)

db = create_engine('sqlite:///somedatabase')
Session = sessionmaker(bind=db)

def get_user(uid):
```

```
    session = Session()
    query = "select * from user where id='%s'" % uid
    return session.execute(query)
```

在这段脚本上运行 Bandit 将检测到三个问题，并详细解释这些问题：

```
$ bandit bandit_example.py
...
Run started:2017-03-20 08:47:06.872002

Test results:
>> Issue: [B404:blacklist] Consider possible security implications
associated with subprocess module.
   Severity: Low Confidence: High
   Location: bandit_example.py:1
1   import subprocess
2   from sqlalchemy import create_engine
3   from sqlalchemy.orm import sessionmaker
--------------------------------------------------
>> Issue: [B506:yaml_load] Use of unsafe yaml load. Allows instantiation
of arbitrary objects. Consider yaml.safe_load().
   Severity: Medium Confidence: High
   Location: bandit_example.py:9


bandit_example.py
8       with open(filename) as f:
9         data = yaml.load(f.read())
10
--------------------------------------------------
>> Issue: [B602:subprocess_popen_with_shell_equals_true] subprocess
call
with shell=True identified, security issue.
   Severity: High Confidence: High
   Location: bandit_example.py:13
12 def run_command(cmd):
13     return subprocess.check_call(cmd, shell=True)
14
--------------------------------------------------
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection
```

```
vector through string-based query construction.
   Severity: Medium Confidence: Low
   Location: bandit_example.py:23
22       session = Session()
23       query = "select * from user where id='%s'" % uid
24       return session.execute(query)
--------------------------------------------------


...
Files skipped (0):
```

本书使用的 Bandit 版本是 1.4.0. 它包含 64 个安全检查，如果想创建自己的检查，通过扩展就能方便地实现。还可通过在项目中创建配置文件，来调整规则。

例如，在调试模式下运行 Flask 时，一个安全检查会因为这是生产环境的安全问题而发出安全警告。请考虑以下示例：

```
$ bandit flask_app.py
...
Test results:
>> Issue: [B201:flask_debug_true] A Flask app appears to be run with
debug=True, which exposes the Werkzeug debugger and allows the execution
of arbitrary code.
   Severity: High Confidence: Medium
   Location: flask_app.py:15
14     if __name__ == '__main__':
15     app.run(debug=True)
```

当准备上线时，这是十分有效的检查，但在开发应用时，你更希望关闭它。一个较好的实践是不对测试模块进行安全扫描。

下面的配置文件可配合 ini 参数来使用，以忽略某些问题以及不扫描 tests/目录下的文件。

```
[bandit]
skips: B201
exclude: tests
```



在持续集成流水线中使用 coveralls 等工具可添加一个 Bandit 调用。如第 3 章所述，这是一种检查代码中是否存在潜在安全问题的绝佳方式。

7.5 本章小结

本章探讨如何在基于微服务的应用环境中使用 OAuth2 和 JWT 令牌进行集中化身份验证和授权。令牌能限制调用者对目标微服务执行的操作，并规定令牌的有效期限。

使用公钥/私钥对，只要令牌发行者未受攻击，还可防止攻击者由于攻入一个服务而破坏整个应用。

除了系统级防火墙规则，Web 应用框架也是防止在 API 上进行欺骗或滥用的好方法，这里感谢 Lua 编程语言的力量，它让我们能方便地通过 OpenResty 实现防火墙。

通过在 Web 服务器(而非 Flask 应用)完成一些事情，OpenResty 成为微服务提速的好方式。

最后，安全的源代码是安全应用的第一步。要遵循良好的编码实践，并确保代码与传入的用户数据和资源进行交互时不做蠢事。虽然诸如 Bandit 的工具不会神奇地让代码变得规范和安全，但它将发现最明显的潜在安全问题。因此，可毫不犹豫地在你的代码库中运行它。

本章未讨论的一个部分是终端用户如何与微服务安全地交互。第 8 章将介绍这部分内容，对所有知识点进行总结，并演示如何通过客户端 JavaScript 应用使用 Runnerly 应用。

第 8 章 综合运用

到目前为止，大部分工作都专注于构建微服务，并让它们彼此交互。现在是时候结合前面介绍的内容来创建用户界面，让最终用户能通过浏览器使用我们的系统。

现代 Web 应用极大地依赖于客户端 JavaScript(JS)。一些 JS 框架一直提供完整的“模型-视图-控制”(Model-View-Controller，MVC)系统，该系统在浏览器中运行并处理文档对象模型(Document Object Model，DOM)。DOM 是浏览器面的结构化表现形式。

Web 开发的范式已从服务器端渲染一切，转变为按需从服务器获取数据，然后在客户端渲染一切。原因是现代 Web 应用动态更改加载的页面的一部分，而不是调用服务器进行完整呈现。这么做速度更快，需要的带宽更少，并提供了更丰富的用户体验。这个转变的最大例子是 Gmail 应用，2004 年左右它是客户端领域的先驱。

与 Facebook 的 ReactJS(https://facebook.github.io/react/)类似的工具提供了高级 API，避免了直接操作 DOM，同时提供了一层抽象，让开发客户端 Web 就像构建 Flask 应用一样方便。

有一种说法，每隔一周就会出现一个新 JS 框架，于是很难决定该使用哪一个。AngularJS(https://angularjs.org/)曾是最酷的玩具，但现在看来，似乎许多开发者已转变为使用普通 ReactJS 来实现应用的大部分 UI。

这种易变性不是坏迹象。它只意味着很多创新发生在 JavaScript 和浏览器生态系统中。一些特性，例如 Service Worker(https://developer.mo zilla.org/en/docs/Web/API/Service_Worker_API)是 Web 开发中的游戏规则改变者；因为它们原生地允许开发者在后台运行 JS 代码。一个新 JS 工具浪潮可能从这个特性中涌现出来。

只要清晰地分离 UI 和系统的其他部分，从一个 JS 框架迁移到另一个就不会太难。这意味着，不应改变微服务发布数据的方式，以使其特定于某个 JS 框架。

对 Runnerly 来说，将用 ReactJS 构建一个小型仪表盘(Dashboard)，然后将其封装在一个专有 Flask 应用中，该应用将表盘与系统其余部分桥接。此外，本章将分析应用如何与微服务交互。

本章包括下面三部分：

- 使用 ReactJS 构建一个仪表盘——ReactJS 简介和示例
- 如何将 ReactJS 嵌入 Flask 应用中
- 身份验证和授权

无论是否选用 ReactJS，在本章的结尾处，你都将理解如何在 Flask 中构建 Web UI，以及如何让其与微服务交互。

8.1 构建 ReactJS 仪表盘

ReactJS 框架实现 DOM 抽象，让所有事件机制变得快捷高效。使用 ReactJS 创建 UI，需要创建包含若干方法的类，当页面创建或更新时，ReactJS 引擎会调用它们。

这个方法意味着，当 DOM 改变时，你不必担心会发生什么。你要做的是实现一些方法，React 将负责完成其余工作。

可使用 JavaScript 或 JSX 来实现 React 中的类，见下一节的讨论。

8.1.1 JSX 语法

JSX 语法扩展(https://facebook.github.io/jsx/)给 JS 添加了 XML 标签，当渲染页面时，诸如 ReactJS 的工具会使用它们。ReactJS 社区将其推广为编写 ReactJS 应用的最佳方式。

在下例中，<script>包含 div 变量，div 变量的值是代表 div 的 XML 树。这个语法是有效的 JSX。然后，ReactDOM.render()函数在 DOM 中渲染 div 变量。

```
<!DOCTYPE html>
<html>
    <head lang="en">
        <meta charset="UTF-8">
    </head>
    <body>
        <div id="content"></div>
        <script src="/static/react/react.min.js"></script>
        <script src="/static/react-dom.min.js"></script>
```

```
        <script src="/static/babel/browser.min.js"></script>

        <script type="text/babel">
          var div =
            <div>
              Hello World
            </div>
          ReactDOM.render(div, document.getElementById('content'));
        </script>
      </body>
    </html>
```

上面两个 ReactJS 脚本是 React 发行版的一部分。browser.min.js 文件是 babel 发行版的一部分，它必须在浏览器遇到任何 JSX 代码前被加载。babel 将 JSX 语法转换成 JS。这个转换称为转译(transpilation)。



babel(https://babeljs.io/)是一个转译器，与其他可用的转译器结合使用，能动态地将 JSX 转换成 JS。只需要将脚本类型标记为 text/babel，即可使用它。

关于 ReactJS，除了 JSX 的特定语法，其他一切都使用常见的 JavaScript 语言。构建 ReactJS 应用时，需要创建用来渲染 Web 页面的 JS 类(可能是 JSX，也可能不是)。

下面分析 ReactJS 的心脏——组件。

8.1.2　React 组件

ReactJS 基于这样的想法：页面可分解为基本组件，在渲染页面的各部分时调用这些组件。

例如，如果想展示跑步活动的列表，可创建 Run 类，从而基于给定值来渲染单一跑步活动；Runs 类遍历跑步活动列表，可调用 Run 类来渲染每一项。

每个类都通过 React.createClass()函数来创建，它接收包含新的类方法的映射。createClass()函数生成新类，通过设置 props 来保存若干属性和传入的方法。

下例在一个新的 JavaScript 文件定义 Run 类。其中 Run 类的 render()函数返回一个 <div>标签：

```
var Run = React.createClass( {
  render: function() {
    return (
      <div>{this.props.title} ({this.props.type})</div>
```

```
        );
      }
    } );

    var Runs = React.createClass( {
      render: function() {
        var runNodes = this.props.data.map(function (run) {
          return (
            <Run
              title= {run.title}
              type= {run.type}
            />
          );
        } );
        return (
          <div>
            {runNodes}
          </div>
        );
      }
    } );
```

Run 类在 div 返回的值是{this.props.title}({this.props.type})，通过访问 Run 实例的 props 属性来渲染。

创建Run实例时会填充props数组——这发生在Runs类的render()方法中。runNodes 变量遍历包含跑步活动的Runs.props.data 列表。

这是最后一块拼图——实例化 Runs 类，以及让 React 使用 props.data 列表来渲染跑步活动。

在 Runnerly 应用中，这个列表可由发布跑步活动的微服务提供。我们可创建另一个 React 类，使用 AJAX(Asynchronous JavaScript And XML)模式和 XmlHttpRequest 类来异步地加载这个列表。

这是下例中的loadRunsFromServer()方法发生的情况。代码使用props里的URL发起GET请求，从服务器获取数据后，通过调用setState()方法设置props.data的值。

```
    var RunsBox = React.createClass( {
      loadRunsFromServer: function() {
        var xhr = new XMLHttpRequest();
        xhr.open('get', this.props.url, true);
```

```
    xhr.onload = function() {
      var data = JSON.parse(xhr.responseText);
      this.setState( { data: data } );
    } .bind(this);
    xhr.send();
  } ,

  getInitialState: function() {
    return {data: []} ;
  } ,

  componentDidMount: function() {
    this.loadRunsFromServer();
  } ,

  render: function() {
    return (
      <div>
        <h2>Runs</h2>
        <Runs data= {this.state.data} />
      </div>
    );
  }
} );

// this will expose RunsBox globally
window.RunsBox = RunsBox;
```

状态(state)的变化会触发 React 类使用新数据来更新 DOM。框架调用 render()方法，该方法会展示包含 Runs 的<div>片段。Runs 实例和每个 Run 实例依次以级联方式传递。

为触发loadRunsFromServer()方法，RunsBox实现了componentDidMount()方法。当创建RunsBox类的实例，将其挂载到React并准备好显示后，会调用componentDidMount()方法。最后，getInitialState()方法会在实例化时被调用，用一个空数组data来初始化props实例。

整个分解和链接过程可能很复杂，但一旦完成，功能将非常强大，因为接下来只需要专注于渲染每个组件即可，React 会用最高效的方式在浏览器中实现其他部分。

每个组件都有一个状态(state)，发生变化时，React 首先更新其内部的 DOM，即虚拟 DOM。一旦虚拟 DOM 发生变化，React 就将必要的变化高效地应用于虚拟 DOM。

本节到目前为止介绍的所有 JSX 代码都可保存为 JSX 模块，并通过以下方式在 HTML 中使用：

```
<!DOCTYPE html>
<html>
  <head lang="en">
    <meta charset="UTF-8">
    <title>Runnerly Dashboard</title>
  </head>
  <body>
    <div class="container">
      <h1>Runnerly Dashboard</h1>
      <br>
      <div id="runs"></div>
    </div>
    <script src="/static/react/react.js"></script>
    <script src="/static/react/react-dom.js"></script>
    <script src="/static/babel/browser.min.js"></script>
    <script src="/static/runs.jsx" type="text/babel"></script>
    <script type="text/babel">
    ReactDOM.render(
      <window.RunsBox url="/api/runs.json" />,
      document.getElementById('runs')
    );
   </script>
  </body>
</html>
```

该示例使用/api/runs.json 来初始化 RunsBox 类。一旦页面加载完毕，React 会调用这个 URL，并期望能得到跑步活动的列表，这个列表被传给 Runs 和 Run 示例。

注意这里使用 window.RunsBox 而非 RunsBox，这是因为 babel 转译器不会在 runs.jsx 文件中公开全局变量。这就是为什么必须设置在 window 的设置属性才能在<script>之间共享的原因。

在浏览器中直接转译是一个糟糕的做法。最好提前转译 JSX 文件，下一节将予以解释。



本节介绍 ReactJS 的基本用法，并未深入探讨所有可能性。要了解更多信息，可阅读它的教程(见 https://facebook.github.io/react/tutorial/tutorial.html)。该教程展示了 React 组件如何通过事件与用户交互——这是掌握了简单渲染方式后的下一步。

现在已经有了构建基于 React 的 UI 的基本布局，接下来分析如何将其嵌入 Flask。

8.2　ReactJS 与 Flask

构建 React 应用时，通常使用 Node.js(https://nodejs.org/en/)来编写服务器端代码。因为使用单一语言并使用相同生态系统的工具会更容易。

尽管如此，React 应用完全可与 Flask 一起运行。可使用 Jinja2 渲染 HTML 页面；与 JavaScript 文件类似，可将转译后的 JSX 文件作为静态文件对外提供服务。此外，如前所述，可使用 React 的 JS 格式的发行版，然后将其与其他文件一起放在 Flask 的静态目录里。

将 Flask 应用命名为 dashboard，可从如下的简单目录结构开始：

- setup.py
- dashboard/
    - __init__.py
    - app.py
    - templates/
        - index.html
    - static/
        - runs.jsx

app.py 文件是一个基本的 Flask 应用，它支持唯一的 HTML 文件：

```
from flask import Flask, render_template,

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

if __name__ == '__main__':
```

```
    app.run()
```

按照 Flask 处理静态文件的惯例，static/目录的所有文件都在/static URL 下。

index.html 模板与上一节描述的模板类似，可变成特定于 Flask 的东西。

以上是从 Flask 运行 ReactJS 应用所需的一切。然而，将 ReactJS 发行版放在 Flask 静态文件库并非维护项目的最佳方式。还需要更好的方式来管理 JS 依赖项。此外，JavaScript 领域有很多不错的工具，下一节会介绍。

8.2.1 使用 bower、npm 和 babel

到目前为止，Flask 应用使用静态 JavaScript 文件来构建 ReactUI。然而，如 JS 社区那样，管理 React 和其他 JavaScript 库的更好方式是将其作为一个可更新的软件包——就像使用 Python 包那样定期更新。为此，可在系统中安装 JavaScript 的包管理器 npm(https://www.npmjs.com/)。npm 通过 Node.js 来安装。在 macOS 上，使用 brew install node；或者，可从 Node.js 主页(https://nodejs.org/en/)下载到系统中。

安装 Node.js 和 npm 后，采用如下方式在命令行使用 npm 命令：

```
$ npm -v
3.5.2
```

为在 Flask 项目中管理 JavaScript 依赖项，将使用 bower(https://bower.io/)。它是 Web 应用的包管理工具，利用 npm 将需要的所有 JS 依赖打包——就像 pip 对 Python 包所做的那样。

使用 npm 来安装 bower：

```
$ npm install -g bower
```

使用-g 开关意味着在系统的 npm 中安装全局 bower。如果安装成功，就得到一个新的 bower 命令行工具。

安装 bower 后，进入 Flask 仪表盘应用的根目录运行交互式 init 命令，如下所示：

```
$ bower init
? name dashboard
? description A ReactJS based Dashboard for Runnerly
? authors Tarek Ziade <tarek@ziade.org>
...

{
   name: 'dashboard',
```

```
    authors: [
        'Tarek Ziade <tarek@ziade.org>'
    ],
    description: 'A ReactJS based Dashboard for Runnerly',
    main: '',
    license: 'MIT',
    homepage: '',
    ignore: [
        '**/.*',
        'node_modules',
        'bower_components',
        'test',
        'tests'
    ]
}

? Looks good? Yes
```

回答几个问题后，这个命令创建名为 bower.json 的配置文件。bower 用它来拉取 JavaScript 库。

由于我们想在 Flask 应用中返回 JavaScript 文件(生成环境中使用 nginx)，还要将静态文件目录的位置告诉 bower。为此，可使用.bowerrc 文件，如下：

```
{"directory": "dashboard/static"}
```

现在，如果运行 bower 的安装命令来安装 React 和 jQuery，会自动处理这两个库，并保存到静态目录中：

```
$ bower install --save jquery react
...
jquery#3.2.1 dashboard/static/jquery
react#15.4.2 dashboard/static/react
```

以上命令将依赖项保存到 bower.json 文件中。重新安装项目时，该机制是非常棒的跟踪依赖的方式。与使用 pip install 命令自动填充 requirements.txt 文件的情形类似。

还需要使用 npm 来安装将 JSX 转译成 JS 文件的 babel 转译器，以及 React 预置插件(preset)，如下所示：

```
$ npm init
$ npm install -save-dev babel-cli babel-preset-react
```

以上命令将软件包安装到当前目录下，并创建与 bower.json 相似的 package.json 文件，此外，node_modules/.bin/目录下也有 babel 命令。

然后，运行下面的命令，将所有 JSX 文件转换成单一的普通 JS 文件，并以 dashboard.js 命名：

```
$ node_modules/.bin/babel dashboard/static/*.jsx >
dashboard/static/dashboard.js
```

一旦运行 babel 命令，Flask 模板可不使用 JSX 文件，而使用位于 JS 文件中的 JS 版本的 React 类。此后，就不需要在客户端动态转译。

同时，这意味着 JSX 文件的所有全局变量在任何地方是可见的，因此，不需要将它们挂载到 window 变量上。

还可将 ReactDOM.render()方法调用(此前位于专属的<script>标签中)移到专属的 zrender.jsx 文件中。

```
ReactDOM.render(
    <RunsBox url="/api/runs.json" />,
    document.getElementById('runs')
);
```

注意文件名以 z 开头，以确保 babel 生成 dashboard.js 时，能将其注入 dashboard.js 的末尾——这是因为 babel 按字母顺序处理这些脚本。这也确保 RunBox 类和其他任何所需的变量或 JS 元素在渲染之前被定义。



还有其他一些方式来处理模块间依赖。诸如 RequireJS(http://www.requirejs.org/)的工具提供了有趣的方式来解决这个问题。然而，现在这个以 Flask 作为后端的小仪表盘不会有大量 JS 文件，当前的方式应该已经足够了。

修改后，最终的 index.html 如下所示：

```
<!DOCTYPE html>
<html>
    <head lang="en">
        <meta charset="UTF-8">
        <title>Runnerly Dashboard</title>
    </head>
    <body>
        <div class="container">
            <h1>Runnerly Dashboard</h1>
            <br>
```

```
            <div id="runs"></div>
        </div>
        <script src="/static/react/react.js"></script>
        <script src="/static/react/react-dom.js"></script>
        <script src="/static/dashboard.js"></script>
    </body>
</html>
```

这一节中，我们始终假定 React 使用位于同一 Flask 应用/api/runs.json 端点下的 JSON 数据。

同一域名下的 AJAX 请求不会有问题，但是，如果需要访问属于其他域的微服务，将需要同时修改服务器端和客户端。

下面分析如何实现跨域访问。

8.2.2 跨域资源共享

允许客户端通过 AJAX 来执行跨域请求具有潜在的安全风险。如果位于域名的客户端页面执行一段 JS 代码，尝试访问不属于你的域名，就可能执行恶意代码并危害用户。

为此，在调用异步请求时，所有浏览器使用同源策略(Same-Origin Policy)，来确保请求发生在同一域名下。

除了安全问题，这也是防止其他应用使用带宽的好方法。例如，站点提供一些字体文件，你可能并不想让其他站点在其页面上使用它们，也不允许不受限地使用你的带宽。

然而，也有一些给其他域名共享资源的使用场景，此时，可在服务上设置一些规则来允许其他域名访问资源。

这就是所谓的跨源资源共享(Cross-Origin Resource Sharing，简称 CORS)。当浏览器向服务发送 AJAX 请求时，会添加 Origin 头，你可判断其是否位于已授权域名的列表中。

如果不在列表中，CORS 协议要求服务器返回包含允许的域名列表的消息头。

还有一个预检机制，浏览器可向端点发送 OPTIONS 方法的请求，来确认发起的请求是否被授权。

在客户端，并不需要关心如何建立这些机制。浏览器会根据请求的具体情况做决定。

但在服务器端，需要确保端点能响应 OPTIONS 请求，并决定允许哪些域名访问资源。如果服务是公开的，可用通配符来允许所有域名。然而，对于一个你控制了客户端的微服务应用，你应该限制请求的域名。

在 Flask 中，使用 flakon 的 crossdomain()装饰器，就能给 API 端点添加 CORS 支持。下面的 Flask 应用中，/api/runs.json 端点能被任何域名使用：

```
from flask import Flask, jsonify
from flakon import crossdomain

app = Flask(__name__)

@app.route('/api/runs.json')
@crossdomain()
def _runs():
   run1 = {'title': 'Endurance', 'type': 'training'}
run2 = {'title': '10K de chalon', 'type': 'race'}
_data = [run1, run2]
return jsonify(_data)

if __name__ == '__main__':
   app.run(port=5002)
```

启动应用，然后通过 curl 来发送 GET 请求，就能看到服务器端添加了 Access-Control-Allow-Origin:* 头：

```
$ curl -v http://localhost:5002/api/runs.json
* Trying localhost...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 5002 (#0)
> GET /api/runs.json HTTP/1.1
> Host: localhost:5002
> User-Agent: curl/7.51.0
> Accept: */*
>
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Type: application/json
< Access-Control-Allow-Origin: *
< Content-Length: 122
```

```
< Server: Werkzeug/0.12.1 Python/3.5.2
< Date: Tue, 04 Apr 2017 07:39:48 GMT
<
[
    {
        "title": "Endurance",
        "type": "training"
    },
    {
        "title": "10K de chalon",
        "type": "race"
    }
]
* Curl_http_done: called premature == 0
* Closing connection 0
```

这是 crossdomain()装饰器的默认许可行为，但你也可为每个端点设置细粒度权限，并将它们限制为特定域名。甚至可使用白名单，只允许指定的请求方法。flakon 也能在 blueprint 级别设置 CORS。

在这里，只允许一个域名已经足够了。例如，假设服务 JS 应用的 Flask 应用运行在 localhost:5000 上，那么可使用以下方式来限制请求的域名：

```
@app.route('/api/runs.json')
@crossdomain(origins=['http://localhost:5000'])
def _runs():
    ...
```

对于不是来自 http://localhost:5000 的请求，服务端不会返回数据。

注意，装饰器会返回 403 响应来拒绝来自不允许的域名的请求。由于 CORS 协议并未定义拒绝访问时应返回的状态码，因此可在实现时自行选择。



要深入理解 CORS，MDN 上有不错的资源，见链接 https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS。

这一节介绍了如何在服务中设置 CORS 头来允许跨域调用，这在 JS 应用中很有用。

为使 JS 应用拥有完整功能，还需要实现身份验证与授权。

8.3 身份验证与授权

React仪表盘需要验证用户的身份，授权其访问某些微服务，以及让用户授予访问Strava的权限。

假定只允许通过验证的用户使用仪表盘。有两类用户：初访使用和回访用户。

初访用户的用户故事如下：

作为一个初访用户，我访问仪表盘时，页面上有“登录”链接。我点击链接时，仪表盘将重定向到 Strava，让我授予仪表盘访问个人资源的权限。然后，Strava 将我重定向到仪表盘，我的信息被关联起来。此后，开始使用我的数据来填充仪表盘。

如上所述，Flask 应用与 Strava 一起运行 OAuth2 Dance 来授权。为与 Strava 关联，需要将访问令牌(access token)存储到 Runnerly 的用户信息中，以便接下来获取跑步活动。

继续深入前，需要做一个设计决定：我们希望将仪表盘与数据服务合并，还是希望将二者分离？

8.3.1 与数据服务交互

第 4 章介绍过，一个设计微服务的安全方法是避免在没有充分理由的情况下创建新服务。

DataService (数据服务)使用数据来保存用户数据，Celery 职程会调用这个服务。首先想到的选项是使用单一 Flask 应用来管理数据库，通过它的 JSON API，不仅给最终用户返回 HTML 和 JS 内容，也服务于其他微服务。

这个方法的好处是，不需要考虑如何实现仪表盘和 DataService 之间的网络交互。此外，除了 ReactJS 应用，不需要在 DataService 添加太多内容就能使其适用于两种情况。

但这样的话，就无法受益于微服务的一个优势，即每个微服务仅关注一个事项。

尽管用保守方法入手总是更安全，但请思考一下拆分对设计的影响。如果仪表盘是独立的，那么需要驱动 DataService 在内部创建和修改用户信息。这意味着 DataService 需要公开若干 API。通过 HTTP 公开数据库的最大风险是，无论何时修改数据库，都可能影响 API。

然而，如果公开的端点尽可能隐藏了数据库结构，就能降低风险。反向做法是公开 CRUD 式 API。

例如，在 DataService 中创建用户时可使用 POST API，只需要将用户的 Strava 令牌和邮箱作为输入，并返回一些用户 ID。这些信息很少改变，仪表盘只需要充当用户和 DataService 之间的代理。

让仪表盘与 DataService 隔离的最大好处在于其稳定性。当构建类似 Runnerly 的应用时，开发者通常进入一个阶段，此时应用的核心部分是稳定的，但会在用户界面和用户体验上进行多次迭代。换句话说，仪表盘可能演进很多，但 DataService 应该很快进入稳定阶段。

根据以上所有理由，将仪表盘和 DataService 作为两个独立应用风险较低。

现在做出了设计决定，下面分析如何与 Strava 执行 OAuth2 Dance。

8.3.2 获取 Strava 令牌

Strava 提供了典型的“三条腿 OAuth2(three-legged OAuth2)”的实现 stravalib (https://github.com/hozn/stravalib)。

先将用户重定向到 Strava，然后公开一个端点。一旦用户授予 Strava 的访问权限，就被重定向到这个调用点。

交换后，就能从用户的 Strava 账户中得到用户信息和令牌的访问权限。可在 Flask 会话中存储这些信息，并当作登录机制，然后将邮箱和令牌的值传给 DataService，以便 Celery 的 Strava 职程使用这个令牌。

如第 4 章所述，以下函数生成了发给用户的 URL：

```
from stravalib.client import Client
def get_strava_url():
    client = Client()
    cid = app.config['STRAVA_CLIENT_ID']
    redirect = app.config['STRAVA_REDIRECT']
    url = client.authorization_url(client_id=cid,
redirect_uri=redirect)
    return url
```

该函数使用 Runnerly 应用的 client_id(在 Strava API 的设置页面生成)，以及为仪表盘定义的重定向地址，然后返回要发给用户的地址。

需要相应地修改仪表盘的视图，将 URL 传给模板。

```
from flask import session

@app.route('/')
```

```
def index():
    strava_url = get_strava_url()
    user = session.get('user')
    return render_template('index.html', strava_url=strava_url,
                           user=user)
```

session 存储 user 时，还传递 user 变量。接下来，模板会使用 Strava URL 来展示登录或登出链接，如下所示：

```
{% if not user %}
<a href="{{strava_url}}">Login via Strava</a>
{% else %}
Hi {{user}}!
<a href="/logout">Logout</a>
{% endif %}
```

用户点击登录链接时，会被重定向到 Strava；返回应用时，会访问 strava_redirect 定义的地址。

视图的实现如下所示：

```
@app.route('/strava_redirect')
def strava_login():
    code = request.args.get('code')
    client = Client()
    cid = app.config['STRAVA_CLIENT_ID']
    csecret = app.config['STRAVA_CLIENT_SECRET']
    access_token = client.exchange_code_for_token(client_id=cid,
client_secret=csecret, code=code)
    athlete = client.get_athlete()
    email = athlete.email
    session['user'] = email
    session['token'] = access_token
    send_user_to_dataservice(email, access_token)
    return redirect('/')
```

stravalib 库的Client类将code换成访问令牌并保存在session中，然后使用get_athlete()方法获取一部分用户信息。

最后，send_user_to_dataservice(email, access_token)方法与微服务 DataService 交互，确保保存了 email 和访问令牌，此时会使用基于 JWT 的访问方式。

由于已在第 7 章中介绍过，这里不再介绍仪表盘与 TokenDealer 交互的细节。过程是类似的——仪表盘应用从 TokenDealer 获取令牌，并用它来访问 DataService。

身份验证的最后一部分在 ReactJS 代码中，见下一节。

8.3.3　JavaScript 身份验证

当仪表盘应用与 Strava 执行 OAuth2 Dance 时，会在 session 中保存用户信息，这是让用户验证仪表盘的完美选择。

然后，当 ReactJS 通过调用 DataService 微服务来展示用户的跑步活动时，需要提供身份验证头。

可采用两个方法来解决该问题：

- 使用现在的 session 信息，通过仪表盘 Web 应用来代理对微服务的所有访问。
- 为每个最终用户生成一个 JWT 令牌，将其保存，在访问其他微服务时使用。

代理方案无疑最简单，因为不需要给每个用户生成令牌来访问 DataService。它还防止公开 DataService。将一切隐藏在仪表盘后的做法，意味着修改应用时有更大灵活性，同时让 UI 保持兼容。

尽管如此，该方法的问题在于强制所有流量经过仪表盘服务，但这并不需要。理想情况下，更新显示的跑步活动列表不应该是仪表盘服务器考虑的事项。

第二个遵循微服务设计的方案更优雅。处理令牌时，Web UI 只是众多微服务的客户端之一。然后，这也意味着客户端不得不处理两个身份验证循环。如果 JWT 令牌被撤销但 Strava 令牌还有效，Client 应用仍需要重新验证身份。

第一个解决方案可能是初版的最佳选择。作为代理代表用户访问微服务，意味着仪表盘应用使用其自身的 JWT 令牌来调用 DataService 以获取用户数据。

如第 4 章所述，DataService 使用以下 API 模式来返回跑步列表：GET /runs/<user_id>/<year>/<month>。

假定仪表盘保存了(e-mail, user ID)元组，针对上面 API 的代理视图可以是 GET /api/runs/<year>/<month>。然后，当用户通过 Strava 登录后，根据 session 中存储的当前用户的电子邮件，仪表盘能找到用户 ID。

代理的代码如下所示：

```
@app.route('/api/runs/<int:year>/<int:month>')
def _runs(year, month):
    if 'user' not in session:
        abort(401)
    uid = email_to_uid(session['user'])
```

```
        endpoint = '/runs/%d/%d/%d' % (uid, year, month)
        resp = call_data_service(endpoint)
        return jsonify(resp.json())
```

call_data_service()函数使用 JWT 令牌访问 DataService 调用点，email_to_uid()函数将电子邮件转换成对应的用户 ID。

最后，为确保上述方法有效，要在每个 xhr 的调用上使用选项 withCredentials，以便发起 AJAX 访问时，同时发出 Cookie 和身份验证头。

```
var xhr = new XMLHttpRequest();
xhr.open('get', URL, true);
xhr.withCredentials = true;
xhr.onload = function() {
    var data = JSON.parse(xhr.responseText);
    ...
} .bind(this);

xhr.send();
```

8.4 本章小结

本章介绍如何将 ReactJS UI 封装到 Flask 应用(仪表盘)。ReactJS 是构建运行在浏览器的现代交互式 UI 的绝佳方式，加快了 JS 的执行，引入了名为 JSX 的新语法。

本章还介绍如何使用基于 npm、bower 和 babel 的工具链来管理 JS 依赖项，以及如何转译 JSX 文件。

仪表盘应用使用 Strava“三条腿 OAuth2”的 API 来关联用户，并从 Strava 服务中获取令牌。我们做了一个设计决定，将仪表盘应用从 DataService 中分离，于是该令牌被发送到微服务 DataService 并存储。以后，Strava Celery 职程会使用该令牌，代表用户抓取跑步活动。

最后，为构建仪表盘，使用“仪表盘”服务器代理对不同微服务的访问，这样简化了客户端的工作。客户端只需要与单一服务器打交道，同时使用单一的身份验证和授权流程。

图 8-1 是新架构，包含仪表盘应用。

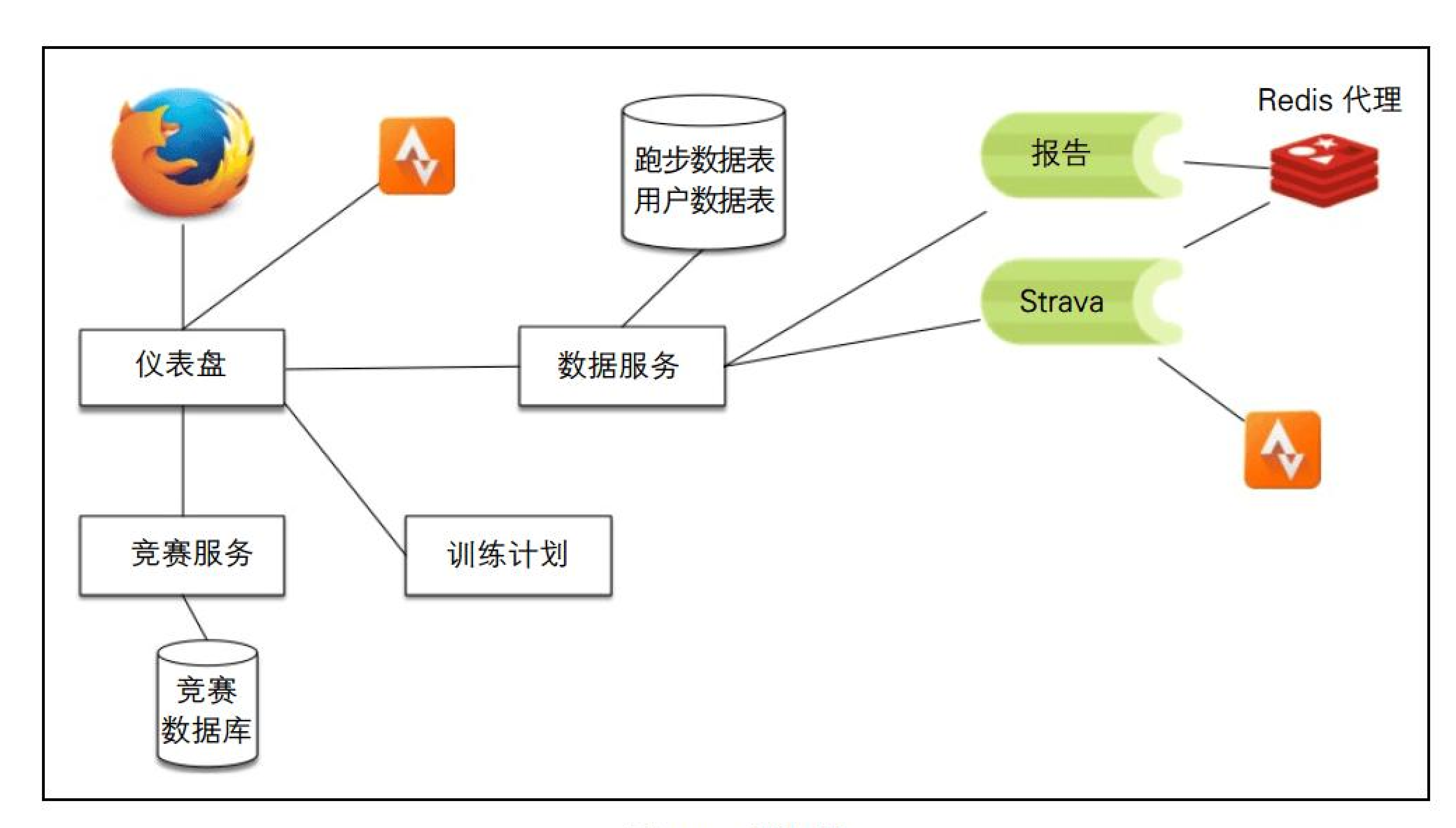


图 8-1　新架构

仪表盘的完整代码位于 Runnerly 组织：https://github.com/runnerly/dashboard。

现在已有了 6 个不同 Flask 应用。作为一个开发者，开发诸如 Runnerly 的应用可能面临挑战。

一个显而易见的需求是，能在单一开发盒子中顺利运行所有微服务。

深入了解 Python 的软件包的工作原理后，第 9 章将介绍如何打包 Python 微服务，以及如何通过进程管理器以开发模式在一个盒子中运行它们。

第 9 章 打包和运行 Runnerly

当 Python 编程语言在 20 世纪 90 年代早期第一次发布时，是通过将代码指向解释器来运行 Python 应用的。关于 Python 项目的打包、发布和分发相关的所有一切都手动完成。当时没有真正的标准，每个项目都有一个冗长的 README 文件，用来描述如何安装所有依赖项。

较大项目使用了系统打包工具来发布——无论是 Debian 软件包、Red Hat Linux 发行版的 RPM 软件包，还是 Windows 的 MSI 软件包。最终，这些项目中的 Python 模块出现在 Python 安装程序下的 *site-packages* 目录中。如果有 C 语言扩展，会放在编译阶段后。

此后，Python 的打包生态系统演进了很多。1998 年，Distutils 被添加到标准库中，为 Python 项目提供基本支持，可创建能安装的分发版本。从那时起，社区中涌现出许多新工具，能改进 Python 项目的打包、发布和分发。

本章将解释如何在微服务上使用最新的 Python 打包工具。

另一个关于打包的热点话题是如何让它匹配日常的开发工作。当构建基于微服务的软件时，要处理很多可移动部件。在特定微服务上工作时，通过使用 TDD 或其他模拟方法(第 3 章介绍过)，大多数情况下可远离这些部署处理。

但是，如果想执行一些接近真实的测试，需要将每个服务都试一试，让整个服务栈都运行在单一盒子中。此外，如果需要随时重新安装微服务的新版本，开发这样一个环境会非常繁杂。

它引出一个问题：如何在环境中正确安装整个微服务栈，并进行开发？

这也意味着如果想使用应用，就必须运行全部微服务。在 Runnerly 中，要打开 6 个不同的 shell 来运行所有微服务，这不应该是开发人员在每次运行应用时都必须要做的事情。

本章将介绍如何利用打包工具从同一环境运行所有微服务。然后通过一个专用进程管理器，只执行一条命令就运行所有微服务。

首先分析如何打包项目，以及打包时应使用哪些工具。

9.1 打包工具链

过去十年中，Python 在打包方面已经演进了许多。人们编写了大量 Python 增强建议(Python Enhancement Proposal，PEP)，这些 PEP 用来改进 Python 项目的安装、发布和分发。

Distutils存在一些缺陷，这使得发布应用较为繁杂。最大的痛点是缺少依赖管理以及处理编译和二进制发布包的方式。所有与编译有关的一切在 20 世纪 90 年代可奏效，但十年后就开始过时了。因为缺乏兴趣，核心团队中没人改进这个库，而且Distutils对于编译Python和大多数项目已足够用了。需要高级工具的人员会使用其他工具，如SCons(http://scons.org/)。

由于很多遗留系统基于 Distutils，因此改善工具链并非易事。从头启动一个新的打包系统相当困难，由于 Distutils 是标准库的一部分，所以引入向后兼容的更改也很难。改进要在新旧版本之间进行。诸如 Setuptools 和 Virtualen 的项目在标准库之外创建，一些更改被直接引入 Python。

即便在今天，依然能找到这些变化留下的疤痕，而且很难确定这一切是如何完成的。例如，pyvenv 命令曾被添加到 Python，后来在 Python 3.6 中又移除了，但 Python 仍然附带虚拟环境模块，在某些方面该模块与 Virtualen 项目存在冲突。

最佳选择是使用那些在标准库以外开发和维护的工具，因为它们的发布周期比 Python 短。换言之，标准库的更改需要几个月才能发布，而第三方项目的发布更快。

所有被认为是事实标准的第三方打包工具链项目现在都在 PyPA(https://www.pypa.io)项目进行分组。

除了开发工具，PyPA 还通过建议 PEP 的方式改进打包标准，并开发早期规范，请参阅 https://www.pypa.io/en/latest/roadmap/。到 2017 年，对于如何打包仍存在疑惑，因为存在多个竞争标准，但问题在整体上得到改善，也许未来会更好。

在开始研究应该使用的工具前，需要讲述一些定义以避免混淆。

9.1.1 一些定义

当我们讨论打包 Python 项目的话题时，对一些术语可能感到困惑，它们的定义随

时间演变，甚至在 Python 领域外它们有不同的含义。

我们需要定义什么是 Python 包、Python 项目、Python 库和 Python 应用。定义如下：

- Python 包(Python package)是包含 Python 模块的目录树。可导入它，它是模块命名空间的一部分。
- Python 项目(Python project)可包含多个包和其他资源，是你要发布的东西。用 Flask 构建的每个微服务都是 Python 项目。
- Python 应用(Python application)是一个可通过用户接口直接使用的 Python 项目。用户接口可以是命令行脚本或 Web 服务器。
- 最后，Python 库(Python library)是一种特定类型的 Python 项目，它提供特定功能供其他 Python 项目使用，但没有直接的用户接口。

应用和库之间的区别可能十分模糊，因为一些库最初是为了给其他项目提供功能的 Python 包，有时也会提供一些命令行工具来公开功能。此外，有时一个库项目也可能变成应用。

为简化流程，最好不对应用和库进行清晰区分。在技术方面唯一的区别是，应用提供更多数据文件和控制台脚本。

现在已经定义了 Python 包、项目、应用和库，下面介绍如何打包项目。

9.1.2　打包

打包 Python 项目时，要有三个必需的文件：

- setup.py：一个特殊模块，用来驱动一切。
- requirements.txt：一个文件，列出所有依赖项。
- MANIFEST.in：一个模板文件，用于列出要包括在发布中的文件。

接下来详细介绍其中每一项。

1. setup.py 文件

setup.py 文件负责管理要与 Python 项目交互的一切信息。执行 setup()方法时，会生成一个符合 PEP 314 格式的元数据文件。这个元数据文件包含项目的所有元数据，要将它放到你使用的 Python 环境中，只能通过调用 setup()重新生成它。

不能使用静态版本的原因是项目的作者可能在 setup.py 中编写与平台相关的代码。根据不同的平台和 Python 版本，它会生成不同的元数据文件。

但通过运行 Python 模块来提取项目的静态信息常出现问题。因为需要确保模块的代码可运行在目标环境的 Python 解释器上。如果想让微服务对开发者社区开放，要注意它可能被安装在不同 Python 环境中。



PEP 390(2009)是第一次抛弃使用 setup.py 生成元数据的尝试。PEP 426、PEP 508 和 PEP 518 是解决这个问题的新尝试，但在 2017 年，我们依然没有支持静态元数据的工具，可能需要很长时间等所有人都使用了新标准后才能出现这样的工具。所以 setup.py 文件还将持续一段时间。

创建 setup.py 文件时的常见错误是：在 setup.py 中导入依赖的包。如果使用诸如 PIP 的工具尝试运行 setup.py 来读取元数据，在列出所有待安装的依赖项前，它可能先抛出导入错误。

唯一可用来在 setup.py 文件中直接导入的依赖项是 Setuptools，因为可假定任何试图安装项目的人都已安装了 Setuptools。

另一个重要考虑因素是描述项目的元数据。虽然只包含名称、版本、URL 和作者项目即可工作，但对描述项目来说，这些信息显然不够。

元数据字段通过 setup()的参数进行设定。有些直接和元数据匹配，有些没有。

以下是微服务项目使用的最小参数集：

- name：包名，是简短的小写字母名称。
- version：项目的版本号，需要符合 PEP 440 定义。
- url：项目的 URL，可以是项目的代码库或主页。
- description：描述项目的一句话。
- long_description：一个 reStructuredText 格式的文档。
- author 和 author_email：作者或组织的姓名和邮箱。
- license：项目使用的许可信息(MIT、Apache2、GPL 等)。
- classifiers：从固定列表中选取的分类器列表，需要符合 PEP 301 定义。
- keywords：描述项目的标签——将项目发布到 Python 包索引(PyPI)时很有用。
- packages：项目包含的包列表——通过 Setuptools 的 find_packages()方法可自动填充该参数。
- install_requires：依赖项列表(一个 Setuptools 参数)。
- entry_points：Setuptools 钩子的列表，如控制台脚本(一个 Setuptools 选项)。
- include_package_data：一个标识，简化了包含的非 Python 文件。
- zip_safe：一个标识，它阻止 Setuptools 将项目安装为 ZIP 文件，是旧标准(可执行的 eggs)。

下面是包含这些选项的 setup.py 文件示例:

```
from setuptools import setup, find_packages

with open('README.rst') as f:
```

```
    LONG_DESC = f.read()

setup(name='MyProject',
      version='1.0.0',
      url='http://example.com',
      description='This is a cool microservice based on strava.',
      long_description=LONG_DESC,
      author='Tarek', author_email='tarek@ziade.org',
      license='MIT',
      classifiers=[
          'Development Status :: 3 - Alpha',
          'License :: OSI Approved :: MIT License',
          'Programming Language :: Python :: 2',
          'Programming Language :: Python :: 3'],
      keywords=['flask', 'microservice', 'strava'],
      packages=find_packages(),
      include_package_data=True,
      zip_safe=False,
      entry_points="""
      [console_scripts]
      mycli = mypackage.mymodule:myfunc
      """,
      install_requires=['stravalib'])
)
```

long_description 参数通常从 README.rst 文件提取，旨在避免在函数中处理一大堆 reStructuredText 格式的字符串。



restructured text-lint 项目(https://github.com/twolfson/restructuredtext-lint)是可用于校验 reStructuredText 文件的语法检测器。

拆分描述信息的另一个好处是：大多数编辑器会自动识别、分析和显示它们。例如，GitHub 将它用作代码库的目录页面，而且提供一个内嵌的 reStructuredText 编辑器，允许直接在浏览器中修改。PyPI 在项目首页也支持类似的显示。

license字段是自由格式，只要人们能识别所使用的许可方式即可。如果使用Apache Public Licence Version 2(APL v2)，就可满足要求。任何情况下，一个作为官方描述许可信息的LICENCE文件应该与setup.py文件放在一起。

classifiers 参数写起来可能最麻烦。需要使用来自 https://pypi.python.org/pypi?%3Aaction=list_classifiers的字符串对项目进行分类。开发人员使用的三个最常见

分类器是支持Python的版本列表、license(和license参数重复而且要保持一致)，以及指示项目成熟度的开发状态。

要将项目发布到 Python 的包索引(Python Package Index，PyPI)上，关键字是让项目显眼的好方法。例如，如果创建一个 Flask 微服务，应使用 Flask 和 microservice 作为关键字。



Trove 分类器是一个机器解析的元数据，可用来与 PyPI 交互。例如，zc.buildout 工具会使用 Framework :: Buildout :: Recipe 分类器来查找包。

entry_points 部分是类似于 INI 的字符串，它定义 Setuptools 的入口点，一旦将项目安装在 Python 中，就可作为插件调用。最常见的入口点类型是控制台脚本。在该部分添加函数时，将在 Python 解释器所在的目录中安装命令行脚本，并通过入口点和函数挂钩。这是一种给项目创建命令行交互界面(Command-Line Interface，CLI)的好方法。在示例中，当项目安装后，可在 shell 中直接使用 mycli 命令行。Python 的 Distutils 具有相似功能，但 Setuptools 做得更好，因为它允许指定特定函数。

最后，install_requires 列出所有依赖项。这个列表包含用到的其他项目，在安装项目的过程中，诸如 PIP 的项目会用到它们。如果这些项目也发布在 PyPI 上，PIP 会找到依赖项然后安装。

一旦创建 setup.py 文件，尝试的一个好方法是创建本地虚拟环境。

假设已安装 virtualen，如果在包含 setup.py 的目录中运行这些命令，它将创建一些目录，其中的 bin 目录包含一个局部 Python 解释器。然后可进入本地 shell。

```
$ virtualen .
$ source bin/activate
(thedir) $
```

这里，运行 pip install -e 命令会在可编辑模式下安装项目。这个命令通过读取 setup 文件来安装项目，但安装过程就地进行。就地安装的含义是，可直接在项目的 Python 模块上进行工作，它们通过 site-packages 目录链接到 Python 的本地安装位置。

使用普通 install 命令会将文件拷贝到本地的 site-packages 目录中，对源代码的改变不影响任何已安装的版本。

PIP 调用还生成 MyProject.egg-info 目录，其中包含元数据。在下例中，PIP 在 PKG-INFO 下生成元数据规范的 1.1 版本。

```
$ more MyProject.egg-info/PKG-INFO
Metadata-Version: 1.1
Name: MyProject
```

```
Version: 1.0.0
Summary: This is a cool project.
Home-page: http://example.com
Author: Tarek
Author-email: tarek@ziade.org
License: MIT
Description: MyProject
        ---------

        I am the **long** description.
Keywords: flask,microservice,strava
Platform: UNKNOWN
Classifier: Development Status :: 3 - Alpha
Classifier: License :: OSI Approved :: MIT License
Classifier: Programming Language :: Python :: 2
Classifier: Programming Language :: Python :: 3
```

这个元数据文件用来描述项目，还用来通过其他命令把项目注册到 PyPI 上。本章后面将详细介绍。

通过在PyPI(https://pypi.python.org/pypi)中查找，PIP调用会拉取所有依赖项，并安装到本地site-packages目录下。要确保一切都符合预期，运行这个命令是个好方法。

我们需要深入讨论 install_requires 参数。它与另一种列出项目依赖项的方法存在冲突，即 requirements.txt 文件，下一节将介绍该文件。

2. requirement.txt 文件

PIP 社区涌现的一个标准是使用 requirements.txt 文件，它列出项目的所有依赖项，还提供扩展语法来安装可编辑的依赖项。请参考 https://pip.readthedocs.io/en/stable/reference/pip_install/#requirements-file-format。

下面是该文件的一个例子：

```
arrow
python-dateutil
pytz
requests
six
stravalib
units
```

社区广泛采纳该文件，因为这样可更容易地记录依赖项。可在项目中创建尽可能多

的 requirements 文件，并允许用户调用 pip install -r thefile.txt 命令来安装其中描述的包。

例如，可以有一个 dev-requirements.txt 文件，其中包含开发阶段的额外工具；或者一个 prod-requirements.txt 文件，其中包含生产环境需要的东西。该格式允许使用继承方式来管理这些 requirements 文件集合。

但使用 requirements 文件会增加一个新问题，它与 setup.py 文件中 install_requires 部分的信息重叠。

为解决该新问题，一些开发者对两者进行区分，一个用来给 Python 库提供依赖信息，另一个用来给 Python 应用提供依赖信息。

在库的 setup.py 文件中使用 install_requires，在用来部署的应用中使用 PIP requirement文件。换句话说，Python应用是不会使用setup.py文件中的install_requires来定义依赖项的。

但这意味着安装应用需要一个特定安装流程，首先使用requirements文件安装依赖项。也意味着对于库类型的项目，将失去使用requirements文件的好处。

前面已介绍过，因为应用和库之间的区别相当模糊，我们不希望由于使用两种不同方式描述 Python 项目依赖关系而导致问题变得复杂。

为避免在这两处重复信息，社区中有一些工具，能自动同步 setup.py 和 requirements 文件的内容。

pip-tools(https://github.com/jazzband/pip-tools)就是其中之一，它通过 pip-complie CLI 生成一个 requirements.txt 文件(或其他文件名)，如下：

```
$ pip install pip-tools
...
$ pip-compile
#
# This file is autogenerated by pip-compile
# To update, run:
#
#    pip-compile --output-file requirements.txt setup.py
#
arrow==0.10.0            # via stravalib
python-dateutil==2.6.0   # via arrow
pytz==2017.2             # via stravalib
requests==2.13.0         # via stravalib
six==1.10.0              # via python-dateutil, stravalib
stravalib==0.6.6
units==0.7               # via stravalib
```

要注意生成的文件包含每个包的版本信息，这称为版本锁定(version pinning)，是根据本地安装的版本填写的。

声明依赖项时，最好在发布项目前锁定所有依赖项。这将确保记录的版本在发布时会被使用和测试。

如果不使用 pip-tools，PIP 有个内置命令 freeze，可用来生成 Python 中安装的所有当前版本的列表。下面是示例：

```
$ pip freeze

cffi==1.9.1
click==6.6
cryptography==1.7.2
dominate==2.3.1
flake8==3.2.1
Flask==0.11.1
...
```

当锁定依赖项时，唯一的问题是其他项目使用相同依赖项但锁定了不同版本。此时 PIP 会抱怨说不能同时满足两个 requirements 集，将无法完成安装。

解决此问题的最简单方法是将未锁定的依赖项放在setup.py文件中，将锁定的放在requirements.txt文件中。该方法可让PIP安装每个包的最新版本。部署时，特别是在生产环境中，可运行pip install -r requirements.txt命令来刷新版本。PIP将升级/降级所有依赖项以匹配版本，当需要时，可在requirements文件中调整它们。

总之，应在每个项目的 setup.py 文件中定义依赖项。只要有一个重现流程，就可根据 setup.py 文件生成 requirements 文件来避免重复，并用 requirements 文件提供锁定的依赖项。

项目中的最后一个必需文件是 MANIFEST.in。

3. MANIFEST.in 文件

创建源码或二进制版本时，Setuptools 将包括所有包模块、数据文件以及 setup.py 文件，还有其他自动包含在 tar 包中的文件。但并不包含诸如 PIP requirements 的文件。

为将这些文件添加到分发版本中，需要添加 MANIFEST.in 文件，其中定义要包含的文件列表。

该文件遵循简单语法，类似于 glob，请参考 https://docs.python.org/3/distutils/commandref.html#creating-a-source-distribution-the-sdist-command。在该文件中指定一个文件或一个目录(包括 glob 模式)，并表示是否要包含或修剪匹配项。

下例来自 Runnerly：

```
include requirements.txt
include README.rst
include LICENSE
recursive-include myservice *.ini
recursive-include docs *.rst *.png *.svg *.css *.html conf.py
prune docs/build/*
```

docs/目录包含 Sphinx 格式的文档，这些问题将被集成在源代码分发版本中，但当构建文档时，在本地 docs/build/生成的任何制品都会被修剪。

一旦有了 MANIFEST.in 文件，在发布项目时，所有文件都应添加到分发版本中。注意可使用 distutils 的 check-manifest 命令来检查文件的语法和有效性。

本书介绍的典型微服务项目包含以下文件：

- setup.py：安装设置文件。
- README.rst：long_description 参数的内容。
- MANIFEST.in：MANIFEST 模板。
- requirements.txt：从 install_requires 生成的 PIP requirements 文件。
- docs/：Sphinx 文档。
- package/：微服务代码的包。

至此，发布项目和创建源代码分发包已经一致了，源代码分发基本也是结构的归档文件。如果包含 C 扩展，还可创建一个二进制分发包。

在讨论如何创建发布前，先介绍如何选择微服务版本号。

9.1.3 版本控制

Python 打包工具不强制执行特定版本模式。version 字段可以是任意字符串。由于每个项目遵循各自的版本模式，这种自由度成为一个问题，有时版本也不兼容安装程序和工具。

要了解版本模式，安装程序需要知道如何进行版本排序和比较，需要解析字符串，并知道一个版本是否比另一个更旧。

早期软件使用的版本方案基于日期，如果软件发布于 2017 年 1 月 1 日，版本号就是 20170101。但如果需要基于分支进行发布，就有问题了。例如，如果软件的版本 2 不向前兼容，那么开始时可能为版本 1 发布一个更新，用来与版本 2 同步，使用日期作为版本号将使版本 1 的发布版看起来比某些版本 2 更新。

有些软件将增量版本和日期合并在一起，但使用日期显然不是处理分支的最佳

方式。

还有很多关于 beta、alpha、release candidate、dev versions 等版本方式的问题。开发者可能希望将一个版本标记成预发布版本。

例如，当 Python 即将推出一个新版本时，会使用 rcX 标记发布一个候选版，这样社区就能在最终版本发布前进行尝试。如 3.6.0rc1、3.6.0rc2 等。

对于一个不发布给社区的微服务，使用这样的标记方式通常有些小题大做了。但当组织以外的人开始使用你的软件时，版本标记就很有用了。

假如要为项目发布一个向后不兼容版本，发布候选版可能有用，在发布前让用户试用它是有好处的。不过，对于通常的发布，使用发布候选版可能适得其反，因为发现问题时，发布新版本的成本很低。



大多数模式下，PIP 做得比较公正，它最终按字母来排序。但如果所有项目都使用相同的版本模式，世界将变得更好。

PEP 386 和 440 的编写试图为 Python 社区提出一个版本模式。它派生自标准的 MAJOR.MINOR[.PATCH]方案，这是一个被开发者广泛采用的方案，对于 pre-和 post-版本有特定规则。

Semantic Versioning(SemVer，http://semver.org/)是社区涌现的另一个标准，除 Python 外的许多地方都在使用它。如果使用 SemVer，只要不使用 pre-release 标记，就能与 PEP440 和 PIP 安装器保持兼容。例如，在 SemVer 中，3.6.0rc2 会转换成 3.6.0-rc2。

与 PEP 440 不同，SemVer 要求提供三版本号。例如，1.0 应该是 1.0.0。

只要移除用来分离版本和标记的破折号，采用 SemVer 就是个好主意。

下面是一个已排序的项目版本列表的示例。都适用于 Python，并与 SemVer 十分接近：

- 9.0
- 0.0a1
- 0.0a2
- 0.0b1
- 0.0rc1
- 0.0
- 1.0

对于微服务项目(或任何 Python 项目)，版本号都从 0.1.0 开始。这样清晰地说明当前是一个不稳定项目，不保证支持向后兼容。在软件足够成熟前，可不断增加 MINOR 版本号。

一旦成熟，常见模式是发布 1.0.0，然后开始遵循以下规则：

- MAJOR 会在为当前 API 引入向后不兼容的更改时递增；
- MINOR 会在添加新功能但不破坏现有 API 时递增；
- PATCH 只在修复 bug 时递增。

当软件处在早期，将该方案严格用于 0.x.x 序列是没意义的。因为会做很多向后不兼容的更改，导致 MAJOR 版本号变成一个大数字。



1.0.0 版本通常是开发者主观控制的。他们希望这是第一个稳定发布版本。因此，当软件被认为稳定时，常从 0.x.x 突然跳到 1.0.0。

对于库，API 是所有公共的文档化功能和类，其他开发者可导入和使用。

对于微服务，代码 API 和 HTTP API 存在区别。你可全面更改微服务项目中的整个实现，但依然使用相同的 HTTP API。要清楚对待这两种 API 版本。

这两种 API 的版本都可遵循此处描述的模式，但一个版本记录在 setup.py(代码)上，一个发布在 Swagger 规范文档中，或用于记录 HTTP API 的任何文档。这两个版本有不同的发布周期。

现在，已经知道如何处理版本号，开始发布吧。

9.1.4 发布

要发布项目，Python 的 Distutils 库中提供了 sdist 命令。

Distutils 有一系列命令可通过 python setup.py <COMMAND>方式来调用。在项目根目录下运行 python setup.py sdist 命令会生成一个包含项目源代码的归档文件。

在下例中，对 Runnerly 的 tokendealer 项目调用 sdist 命令：

```
$ python setup.py sdist
running sdist
[...]
creating runnerly-tokendealer-0.1.0
creating runnerly-tokendealer-0.1.0/runnerly_tokendealer.egg-info
creating runnerly-tokendealer-0.1.0/tokendealer
creating runnerly-tokendealer-0.1.0/tokendealer/tests
creating runnerly-tokendealer-0.1.0/tokendealer/views
copying files to runnerly-tokendealer-0.1.0...
copying README.rst -> runnerly-tokendealer-0.1.0
[...]
copying tokendealer/tests/__init__.py -> runnerlytokendealer-
0.1.0/tokendealer/tests
```

```
copying tokendealer/tests/test_home.py -> runnerlytokendealer-
0.1.0/tokendealer/tests
copying tokendealer/views/__init__.py -> runnerlytokendealer-
0.1.0/tokendealer/views
copying tokendealer/views/home.py -> runnerlytokendealer-
0.1.0/tokendealer/views
Writing runnerly-tokendealer-0.1.0/setup.cfg
creating dist
Creating tar archive
removing 'runnerly-tokendealer-0.1.0' (and everything under it)
```

sdist 命令从 setup.py 和 MANIFEST.in 文件读取信息，然后抓取所有文件并存放到归档文件中。在 dist 目录中创建结果。

```
$ ls dist/
runnerly-tokendealer-0.1.0.tar.gz
```

注意归档文件名包含项目名和版本。该存档可直接使用 PIP 来安装，安装时使用如下方式：

```
$ pip install dist/runnerly-tokendealer-0.1.0.tar.gz
Processing ./dist/runnerly-tokendealer-0.1.0.tar.gz
  Requirement already satisfied (use --upgrade to upgrade):
runnerlytokendealer==
0.1.0 [...]
Successfully built runnerly-tokendealer
```

当没有需要编译的扩展时，源码发布已经足够了。如果需要编译，目标系统有必要在安装时再编译一次。这意味着目标系统需要一个编译器，但这种场景并不常见。

另一种方式是预编译源码，然后给每个目标系统创建二进制分发版。Disutils 有很多 bdist_xxx 命令进行二进制分发，但这些命令已不再维护了。新方法是在 PEP427 中定义 Wheel 格式。Wheel 格式是一个 ZIP 文件，包含部署到目标系统需要的所有文件，不要求在安装时重新运行命令。

如果项目没有 C 扩展，那么发布 Wheel 分发版很有益处。因为安装过程比 sdist 更快；此时 PIP 只是移动文件，不需要运行任何命令。

要构建 Wheel 归档文件，需要安装 wheel 项目，然后调用 bdist_wheel 命令(此命令将在 dist 目录创建一个新的存档文件)。

```
$ pip install wheel
```

```
$ python setup.py bdist_wheel --universal
$ ls dist/
runnerly-tokendealer-0.1.0.tar.gz
runnerly_tokendealer-0.1.0-py2.py3-none-any.whl
```

注意示例在调用 bdist_wheel 时使用了-universal 标识。

如果项目兼容 Python 2 和 3，该标识意味着命令生成的源代码发布包可同时在两个环境安装，安装时不需要额外步骤(如 2 到 3 的转换)。如果没有该标识，在安装时会创建 runnerly_tokendealer-0.1.0-py2.py3-none-any.whl 文件，告知该发布包只能用在 Python 3 上。

如果有 C 扩展，bdist_wheel 会检测到它，然后创建一个平台特定的分发版，该分发版包含已编译的扩展。这种情况下，文件名中的 none 被替换成平台名。

如果 C 扩展未绑定到特定系统库上，创建平台特定发布包是没问题的。如果有绑定，二进制发布包可能无法在任何系统上工作，尤其当目标系统有相同库的不同版本时。发布一个能在所有环境都正常工作的二进制发布包是很难的。有些项目会将使用的所有扩展库通过静态连接打包在一起。通常，在写一个微服务时很少使用 C 扩展，所以源代码分发包已经足够了。

发布 sdist 和 Wheel 分发是最佳实践。诸如 PIP 的安装器会选择 wheel，项目在安装时比 sdist 发布包更快。换句话说，sdist 发布包可用在较旧的安装器中，或用于手动安装。

一旦归档文件准备就绪，即可考虑如何分发它们。

9.1.5 分发

开发开源项目时，将项目发布到 PyPI(https://pypi.python.org/pypi.)是很好的实践。

与很多现代编程语言生态一样，安装器会浏览索引并查找发布版，然后下载。

当调用 pip install <project>命令时，PIP 会浏览 PyPI 索引查找项目是否存在，以及是否有适合平台的分发包。

公开的名称是在 setup.py 文件中使用的名称，需要将它注册到 PyPI 以便能公开分发包。索引遵循“先到先服务”原则，所以如果名称被占用，就需要换一个。

为应用或组织创建微服务时，可给项目添加公共前缀。Runnerly 应用使用 runnerly 作为前缀。

在包级别上，前缀有时也有助于避免冲突。

Python 有一个包名空间功能，允许创建顶级包名(如 runnerly)，然后将多个项目最终安装在顶级 runnerly 包下。

其效果是，当导入时，每个包都获得公用的 runnerly 名称空间，这是将代码分组到同一标记下的优雅方法。可通过标准库中的 pkgutil 模块使用该功能。

为使用该功能，只需要在每个项目中创建同样的顶级目录，通过在__init__.py 包含所有绝对导入，将顶级名称用作前缀。

```
from pkgutil import extend_path
__path__ = extend_path(__path__, __name__)
```

举个例子，在 Runnerly 中，如果我们决定用同一名称空间发布所有项目，每个项目有相同的顶级包名。tokendealer 可像下面这样：

- runnerly
    - __init__.py:包含 extend_path 调用
    - tokendealer/
        - ..实际代码

dataservice 可像下面这样：

- runnerly
    - __init__.py：包含 extend_path 调用
    - dataservice/
        - ..实际代码

两者都将发布 runnerly 顶级包，当 PIP 安装它们时，tokendealer 和 dataservice 包最终安装在相同的目录(site-packages/runnerly)中。

生产环境中的每个微服务都独立安装，该功能在生产环境没多大用处，但也没坏处；另外，如果创建许多跨项目使用的库，该方法将非常有用。

到此，我们假设每个项目都是独立的，每个名称在 PyPI 中都是可用的。

要在 PyPI 上发布，需要使用该页面(https://pypi.python.org/pypi?%3Aaction=register_form)的表单注册一个新用户，如图 9-1 所示。

Manual user registration

This form allows "traditional" registration (using a password). Users who want to register with their OpenID (e.g. Google or Launchpad account) should follow one of the links to the right.

You can use your PyPI account to log into other services supporting OpenID. You need to first log into PyPI before logging into other services (doing it the other way is prone to phishing attacks). To log in, simply type **pypi.python.org** into the field asking for an OpenID. Your OpenID is **https://pypi.python.org/id/**; you can also use this ID directly to log in.

Username:
Password:
Confirm:
Email Address:
PGP Key ID (optional): (This identifies a PGP or GPG key)
Register

图 9-1　注册一个新用户

一旦有了用户名和密码，就可在主目录创建一个包含凭证信息的.pypirc 文件，像下面这样：

```
[pypi]
username = <username>
password = <password>
```

每次与 PyPI 索引交互时，会用该文件创建一个包含基本身份验证的消息头。

Python Distutils 有注册和上传命令，用来在 PyPI 上注册新项目，但 Twine(https://github.com/pypa/twine)更好用，用户界面略好一些。

一旦安装 Twine(使用 pip install twine 命令)，就可使用以下命令来注册包：

```
$ twine register dist/runnerly-tokendealer-0.1.0.tar.gz
```

该命令将使用包中的元数据信息在 PyPI 创建一个新条目。

一旦完成，就可使用下面的命令上传分发文件：

```
$ twine upload dist/*
```

这样，你的包应该上传到 PyPI，https://pypi.python.org/pypi/<project>有一个 HTML 主页。可使用 pip install <project> 命令进行安装。

现在，已经知道如何打包每个微服务，为让开发更便捷，接下来分析如何在同一环境中运行它们。

9.2 运行所有微服务

可使用内置的 Flask Web 服务器来运行微服务。通过该脚本运行 Flask 应用时需要设置一个环境变量，该环境变量指向包含 Flask 应用的模块。

在下面的 Runnerly 应用例子中，dataservice 微服务位于应用的 runnerly.dataservice 模块中，可用以下命令从根目录运行 dataservice。

```
$ FLASK_APP=runnerly/dataservice/app.py bin/flask run
 * Serving Flask app "runnerly.dataservice.app"
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
127.0.0.1 - - [01/May/2017 10:18:37] "GET / HTTP/1.1" 200 -
```

使用 Flask 命令行运行应用是很好的方式，但限制只能使用它的接口参数。如果需要使用很多参数来运行微服务，就必须添加环境变量。

另一个方法是使用 argparse 模块(https://docs.python.org/3/library/argparse.html)创建自己的运行器，给每个微服务添加任何参数。

下例是一个可工作的运行器，将基于 argparse 的命令行脚本运行 Flask 应用。它接受单一参数--config-file，该文件包含运行微服务所需的一切配置。

```
import argparse
import sys
import signal
from .app import create_app

def _quit(signal, frame):
    print("Bye!")
    # add any cleanup code here
    sys.exit(0)

def main(args=sys.argv[1:]):
    parser = argparse.ArgumentParser( description='Runnerly
                                      Dataservice')
    parser.add_argument('--config-file', help='Config file',
                        type=str, default=None)
    args = parser.parse_args(args=args)

    app = create_app(args.config_file)
    host = app.config.get('host', '0.0.0.0')
    port = app.config.get('port', 5000)
    debug = app.config.get('DEBUG', False)
    signal.signal(signal.SIGINT, _quit)
    signal.signal(signal.SIGTERM, _quit)
    app.run(debug=debug, host=host, port=port)

if __name__ == "__main__":
    main()
```

该方法提供很大的灵活性。为使该脚本成为控制台脚本，需要通过 entry_points 参数将其传递给 setup 类的函数，如下所示：

```
from setuptools import setup, find_packages
from runnerly.dataservice import __version__

setup(name='runnerly-data',
```

```
        version=__version__,
        packages=find_packages(),
        include_package_data=True,
        zip_safe=False,
        entry_points="""
        [console_scripts]
        runnerly-dataservice = runnerly.dataservice.run:main
        """)
```

通过该参数，可创建一个 runnerly-dataservice 控制台脚本，并将其链接到前面的 main()函数上。

```
$ runnerly-dataservice --help
usage: runnerly-dataservice [-h] [--config-file CONFIG_FILE]

Runnerly Dataservice

optional arguments:
  -h, --help show this help message and exit
  --config-file CONFIG_FILE
                        Config file

$ runnerly-dataservice
 * Running on http://127.0.0.1:5001/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger pin code: 216-834-670
```

此前在 PIP 中使用了-e 参数，以便在开发模式下运行项目。如果在同一个 Python 下对所有微服务都用相同参数，就能在同一环境中使用各自的运行器运行它们。

你可创建新的 virtualen，然后用-e 在 requirements.txt 文件中链接每个开发目录，requirements.txt 文件会列出所有微服务。

PIP 可识别出 Git URL，并在环境中克隆代码库，这样创建一个包含所有代码的根目录将变得非常方便。

例如，下面的 requirements.txt 文件指向两个 GitHub 仓库：

```
-e git+https://github.com/Runnerly/tokendealer.git#egg=runnerly-
tokendealer
-e git+https://github.com/Runnerly/data-service.git#egg=
runnerly-data
```

这时，运行 pip install -r requirements.txt 命令将两个项目克隆到 src 目录，然后在开发模式下安装它们，这意味着可直接在 src/目录进行修改并提交代码。

最后，假设在运行微服务所需之处都创建了控制台脚本，这些脚本将添加到 virtualen 的 bin 目录中。

运行微服务难题的最后一部分是避免在单独 bash 窗口中运行每个控制台脚本。我们希望用一个脚本来管理这些进程。下一节将介绍如何使用进程管理器来完成此操作。

9.3　进程管理

第 2 章介绍过，基于 Flask 的应用通常在一个单线程环境中运行。

要增加并发，最常见的模式是使用预派生模式(prefork mode)。通过派生若干个进程(称为职程)来同时服务于多个客户，这种方式能在同一个套接字上接收多个传入连接。套接字可以是 TCP 套接字或 Unix 套接字。当客户端和服务器都在同一台计算机上运行时，可使用 Unix 套接字；因为它们基于文件来交换数据，不必处理网络开销，因此比 TCP 套接字稍快。当应用通过诸如 nginx 的前端服务器代理请求时，常见做法是使用 Unix 套接字运行 Flask 应用。

无论 Unix 还是 TCP，每当一个请求到达套接字时，会由第一个可用进程接收并处理。至于哪个进程获取哪个请求，是由系统套接字 API 完成的，这些 API 在系统层面带有锁机制。这种轮询机制能在所有进程之间对请求进行负载均衡，而且非常有效。

为在 Flask 应用使用该模式，可使用 uWSGI(http://uwsgi-docs.readthedocs.io)。通过设置 processes 参数来预派生多个进程，然后服务 Flask 应用。

uWSGI 工具很好用，有多个参数。甚至还有通过 TCP 通信的自有二进制协议。对于给 Flask 应用提供服务来说，在 nginx HTTP 服务器后运行自有二进制协议的 uWSGI 是不错的方案。uWSGI 负责管理进程，同时与 nginx 或其他 HTTP 代理进行交互，或直接与终端用户交互。

然而，uWSGI 是一个专门运行 Web 应用的工具。为部署一个开发环境，其他进程(如 Redis 实例)需要与微服务在同一环境运行，这时需要使用另一个进程管理器。

一个较好的进程管理器方案是 Circus(http://circus.readthedocs.io)，它可运行任何类型的进程，即便不是一个 WSGI 应用，它也能绑定一个套接字进行进程管理。换句话说，Circus 可运行有多个进程的 Flask 应用，也可管理其他进程。

Circus 是一个 Python 应用，所以通过 pip install circus 命令来安装它。一旦安装完成，就会获得很多新命令。两个最基本的命令是 circusd 和 circusctl，前者是一个进程

管理器，后者通过命令行操控进程管理器。

Circus 使用类似 INI 格式的配置文件，可在其中的特定部分列出要运行的命令，并在每一部分指定需要的进程数量。

Circus 也能绑定套接字，通过文件描述符让派生的进程使用它们。在系统上创建套接字时，会使用一个文件描述符(File Descriptor，FD)。FD 是程序用来访问文件或 I/O 资源(比如套接字)的系统句柄。从另一个进程派生的进程将继承所有文件描述符。通过该机制，由 Circus 启动的所有进程都可共享相同的套接字。

下例运行两个命令，一个为 server.py 模块的 Flask 应用运行 5 个进程，一个运行 Redis 服务器进程。

```
[watcher:web]
cmd = chaussette --fd $(circus.sockets.web) server.application
use_sockets = True
numprocesses = 5

[watcher:redis]
cmd = /usr/local/bin/redis-server
use_sockets = False
numprocesses = 1

[socket:web]
host = 0.0.0.0
port = 8000
```

socket:web 部分描述用来绑定 TCP 套接字的主机和端口，watcher:web 部分通过 $(circus.sockets.web)变量进行引用。当 Circus 运行时，会使用套接字的文件描述符的值来替换变量。

使用 circusd 命令行运行该脚本：

```
$ circusd myconfig.ini
```

有一些 WSGI的Web服务器提供运行特定文件描述符的选项，但大多数服务器并不公开这些参数，它们会在给定主机和端口绑定一个新的套接字。

Chaussette(http://chaussette.readthedocs.io/)可让你通过一个 FD 来运行现有的大多数 WSGI Web 服务器。执行 pip install chaussette 命令后，就可运行各种后端 Flask 应用了，这些应用位于 http ://chaussette.readthedocs.io/en/latest/#backends 页面的列表里。

对我们的微服务，使用 Circus 意味着可通过 circusd 命令给每个服务创建一个观察器和套接字部分。

唯一区别是，如果使用自己的启动器而不是 Chaussette 控制台，需要对其进行调整，以便使用文件描述符来运行。

下面的例子中，当使用--fd 选项启动微服务时，main()函数会使用 Chaussette 的 make_server()函数：

```
from chaussette.server import make_server

def main(args=sys.argv[1:]):
    parser = argparse.ArgumentParser(description='Runnerly
                                     Dataservice')
    parser.add_argument('--fd', type=int, default=None)
    parser.add_argument('--config-file', help='Config file',
                        type=str, default=None)
    args = parser.parse_args(args=args)
    app = create_app(args.config_file)
    host = app.config.get('host', '0.0.0.0')
    port = app.config.get('port', 5000)
    debug = app.config.get('DEBUG', False)
    signal.signal(signal.SIGINT, _quit)
    signal.signal(signal.SIGTERM, _quit)

    def runner():
        if args.fd is not None:
            # use chaussette
            httpd = make_server(app, host='fd://%d' % args.fd)
            httpd.serve_forever()
        else:
            app.run(debug=debug, host=host, port=port)

    if not debug:
        runner()
    else:
        from werkzeug.serving import run_with_reloader
        run_with_reloader(runner)
```

然后，在 circus.ini 文件中进行配置：

```
[watcher:web]
cmd = runnerly-dataservice --fd $(circus.sockets.web)
use_sockets = True
```

```
numprocesses = 5

[socket:web]
host = 0.0.0.0
port = 8000
```

有了这些，如果需要调试一个特定微服务，那么常用模式是在将要调用的 Flask 视图中增加一个 pdb.set_trace()调用。

一旦在代码中添加该调用，就可通过 circusctl 停止微服务，然后在另一个 shell 中手动运行它，这样就能启用调试模式了。



circus 还提供将 stdout 和 stderr 流重定向到日志文件的功能，以便给调试和其他许多功能提供方便，相关信息可在 https://circus.readthedocs.io/en/latest/for-ops/configuration/找到。

9.4 本章小结

本章介绍如何打包、发布和分发每个微服务。目前 Python 打包仍需要使用一些遗留工具，在 Python 和 PyPA 成为主流前，这种情况将持续数年。

但只要有了标准的、可重复的、文档化方式对微服务进行打包和安装，就已经够用了。

运行一个应用可能包含很多项目，这会增加部署复杂度。因此在同一个环境中运行所有东西变得非常重要。

诸如 PIP 的开发模式和诸如 Circus 的工具对这种场景很有用，能帮助简化运行整个栈的工作，但即便这些都在一个 virtualenv 中，仍需要在系统中安装一些东西。

在环境中运行所有东西的另一个问题是，使用的操作系统可能不是生产环境的操作系统，或为其他目的安装的库会造成干扰。

防止该问题的最佳方式是在虚拟环境中完全独立地运行整个应用栈。这是下一章将讨论的内容，例如如何在 Docker 内运行服务。

第 10 章 容器化服务

前一章中，我们直接在宿主机的操作系统上运行不同微服务。因此，应用需要的所有依赖项和数据都直接安装到操作系统上。

这种做法在大部分情况下是可行的。因为通过虚拟环境(virtualenv)运行 Python 应用时，会将依赖项下载并安装到单独目录中。但若应用使用了数据库系统，那么需要在操作系统上运行数据库(除非只使用 SQLite 文件)。对某些 Python 库，系统中可能需要包含某些头文件来编译扩展。

系统很快就会安装和使用多种软件。在开发环境下，如果不需要在不同版本的服务上工作，这样做就不会有问题。然而，如果其他一些潜在贡献者试图在本地安装应用，就必须在系统级别安装许多软件，这破坏了与贡献者的默契。

此时，使用虚拟机(VM)是运行应用的一个好方案。过去十年里，许多软件项目只有精心配置才能运行，因此它们通过 VMWare 或 VirtualBox 等工具提供了可立即运行的虚拟机。这些虚拟机包含完整运行栈，如预置的数据库。只需要一个命令，就可在大多数平台上轻松运行演示程序。

然而，其中一些工具并不完全开源，它们运行起来非常慢，而且会贪婪地使用内存和 CPU，同时磁盘的 I/O 也很糟。很难想象在生产环境中运行它们，因此通常只在演示中使用。

Docker 带来一场革命。它是一个开源的虚拟化工具，于 2013 年第一次发布，此后变得非常流行。此外，与 VMWare 或 VirtualBox 不同，Docker 能在生产环境中以原生速度运行应用。

换言之，为应用创建镜像(image)不仅是为了演示或本地开发，镜像可用在真正的部署环境中。

本章将介绍 Docker，并解释如何用它来运行基于 Flask 的微服务。然后介绍 Docker

生态系统中的一些工具。最后介绍集群。

10.1 何为 Docker?

Docker(https://www.docker.com/)项目是一个容器平台，它允许应用运行在隔离环境中。Docker 利用现有 Linux 技术(如 cgroups，https://en.wikipedia.org/wiki/Cgroups)提供一组高级工具，来驱动一系列正在运行的进程。由于 Linux Kernel 是必要条件，因此在 Windows 和 macOS 上，Docker 需要与 Linux 虚拟机交互才可运行。

作为Docker用户，只需要指定希望运行的镜像，Docker就能通过与Linux 内核的交互来完成所有繁重工作。此上下文中的“镜像”指为了运行容器在Linux内核上运行一组进程所需指令的总和。镜像包括运行Linux发行版所需的一切资源。例如，如果宿主机的操作系统不是Ubuntu发行版，仍可在Docker容器中运行某个Ubuntu版本。



尽管在 Windows 上使用 Docker 是可能的，但 Flask 微服务始终应该部署在 Linux 或基于 BSD 的系统上——本章剩余部分都基于如下假设：所有一切都安装在诸如 Debian 的 Linux 发行版上。

如果已在第 6 章中安装 Docker 并配置了 Graylog 实例，那么可直接阅读下一节“Docker 简介”。

如果尚未安装 Docker，可访问 https://www.docker.com/get-docker 中的 Get Docker 部分。社区版 Docker 对于构建、运行和安装容器来说已经够用了。在 Linux 上安装 Docker 是傻瓜式的——有可能在 Linux 发行版上找到 Docker 的软件包。

对于macOS来说，Docker使用虚拟机来运行Linux内核。最新版本基于HyperKit (https://github.com/moby/hyperkit)并利用bhyve(一个BSD Hypervisor)。通过虚拟机来运行Docker会增加一些开销，但它的量级轻，能运行在现代硬件上。在主流操作系统上，Hypervisor正变得常见。

在 Windows 上，Docker 使用 Windows 内置的 Hyper-V 技术(可能需要启动它，才可使用)。通常可在命令行中使用 DSIM 调用来启动，如下：

```
$ DISM /Online/Enable-Feature/All/FeatureName:Microsoft-Hyper-V
```

如果安装成功，可在命令行中运行 docker 命令。可尝试使用 version 命令来验证是否安装成功：

```
$ docker version
Client:
```

```
Version:        17.03.1-ce
API version:    1.27
Go version:     go1.7.5
Git commit:     c6d412e
Built:          Tue Mar 28 00:40:02 2017
OS/Arch:        darwin/amd64

Server:
Version:        17.03.1-ce
API version:    1.27 (minimum version 1.12)
Go version:     go1.7.5
Git commit:     c6d412e
Built:          Fri Mar 24 00:00:50 2017
OS/Arch:        linux/amd64
Experimental:   true
```

Docker 的安装由 Docker 服务器和 Docker 客户端组成。前者是守护进程执行的引擎，后者是命令行程序(例如 Docker)。

服务器提供 HTTP API，可使用本机的 Unix 套接字(通常是/var/run/docker.sock)或使用网络来访问。

换句话说，Docker 客户端能与运行在其他机器上的 Docker 守护进程交互。



Docker 命令行工具对于手动管理 Docker 来说已经很好了。但若需要编写脚本来操作 Docker，那么诸如 docker-py(https://github.com/docker/docker-py)的 Python 库允许使用 Python 做任何事情。它使用 requests 库对 Docker 守护进程执行 HTTP 调用。

安装 Docker 后，来分析它如何工作。

10.2 Docker 简介

在 Docker 中运行容器是通过执行一系列命令完成的，这些命令启动一组进程，将容器与系统其余部分隔离。

可使用 Docker 来运行单一进程，但在实践中，我们要通常期望运行完整的 Linux 发行版。但不必担心，Docker 已经提供了运行完整 Linux 所需的一切。

目前所有 Linux 发行版都提供基础镜像(base image)，使用基础镜像就可在 Docker 中运行对应的发行版。使用镜像的一个典型方式是创建一个 Dockerfile(见文档

https://docs.docker.com/engine/reference/builder/)。在这个文件中，首先指向期望使用的基础镜像，然后添加创建容器所需的其他命令。

下面是 Dockerfile 的一个简单示例：

```
FROM ubuntu
RUN apt-get update && apt-get install -y python
CMD ["bash"]
```

Dockerfile 是一个包含一系列指令的文本文件。每一行以大写字母的指令开始，紧跟着它的参数。

上例包括三条指令：

- FROM：指向要使用的基础镜像。
- RUN：在基础镜像安装完毕后，在容器中运行命令。
- CMD：当 Docker 执行容器时运行的命令。

在 Dockerfile 文件所在的目录中使用 Docker 的 build 和 run 命令，就能创建并运行镜像。注意末尾的点号(.)。

```
$ docker build -t runnerly/python .
Sending build context to Docker daemon 6.656 kB
Step 1/3 : FROM ubuntu
 ---> 0ef2e08ed3fa
Step 2/3 : RUN apt-get update && apt-get install -y python
 ---> Using cache
 ---> 48a5a722c81c
Step 3/3 : CMD bash
 ---> Using cache
 ---> 78e9a6fd9295
Successfully built 78e9a6fd9295
$ docker run -it --rm runnerly/python
root@ebdbb644edb1:/# python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

以上代码片段中的-t 选项给镜像添加一个标签。上面的例子中，镜像被打上 runnerly/python 标签。将项目或组织名作为前缀是标签的一个惯例，这样做便于给镜像分组，并将它们放在相同的名称空间下。

Docker 创建镜像时也创建一个缓存，包含 Dockerfile 中的每条指令。在不修改

Dockerfile 的前提下，如果再次运行 build 命令，就能快速构建镜像。交换或修改指令会导致从第一个改动处开始重建镜像。出于这个原因，编写 Dockerfile 的一个策略是对命令进行排序，将最稳定的命令(几乎不会修改的那些)放在最前面。

Docker 的一个重大特性是提供与其他开发者一起共享、发布和重用镜像的能力。Docker Hub(https://hub.docker.com)之于 Docker 镜像，就如 PyPI 之于 Python 软件包。

在上例中，Docker 从 Docker Hub 中拉取基础镜像 ubuntu。Docker Hub 中有许多预先存在的镜像供你使用。

例如，若想启动一个针对 Python 调整的 Linux 发行版，可到官方 Docker Hub 的 Python 主页挑选(见 https://hub.docker.com /_/python/)。

形如 python:version 的镜像基于 Debian，对于任何 Python 项目，都是不错的起点。

基于 Alpine Linux(参见 http://gliderlabs.viewdocs.io/docker-alpine/)的 Python 镜像也非常流行，是运行 Python 的最小镜像。它们比其他镜像小十倍以上，这意味着对于想在 Docker 中运行项目的人而言，可快速地下载和准备镜像。

为在 Alpine 中运行 Python 3.6，可创建如下 Dockerfile 文件：

```
FROM python:3.6-alpine
CMD ["python3.6"]
```

构建并运行这个 Dockerfile 后，会启动 Python 3.6 的交互式 shell(需要添加-it 参数)。如果 Python 应用不依赖系统级别的依赖项，也不需要执行编译任务，那么 Alpine 系列的镜像就很好用。Alpine 有一组包含编译工具的特殊镜像，但有时这些工具与一些项目不兼容。

对于基于 Flask 的微服务应用来说，基于 Debian 的镜像或许是简单的选择。这是因为它包含标准编译环境，且比较稳定。此外，一旦下载基础镜像，它就会被缓存和复用，因此不要再次下载所有东西。



由于任何人都可向 Docker Hub 上传镜像，所以要注意，务必使用可信的人和组织的镜像。使用镜像时，除了有运行恶意代码的风险外，另一个问题是 Linux 镜像可能没有打上最新安全补丁。

10.3　在 Docker 中运行 Flask

为在 Docker 中运行 Flask，可使用基础 Python 镜像。

接下来，可通过 PIP 来安装应用及其依赖项。而 PIP 已安装在 Python 镜像中。

假定项目使用 requirements.txt 文件来定义依赖项，并使用 setup.py 文件安装项目，

那么可让 Docker 使用 pip 命令给项目创建镜像。

下例添加两个指令——COPY 命令递归地将一个目录结构拷贝至 Docker 镜像中，RUN 命令在命令行中执行 PIP：

```
FROM python:3.6
COPY . /app
RUN pip install -r /app/requirements.txt
RUN pip install /app/
EXPOSE 5000
CMD runnerly-tokendealer
```

这里的标签 3.6 会从 Docker Hub 获取最新 Python 3.6 镜像。

COPY 命令自动在容器的根目录创建 app 目录，并将当前目录(.)中的所有内容复制到其中。COPY 命令的一个重要细节是，对当前目录(.)的任何修改都会让 Docker 的缓存失效，导致下一次构建时从这一步开始。

要调整这个机制，可创建.dockerignore 文件，列出那些需要让 Docker 忽略的目录或文件。

下面构建这个 Dockerfile:

```
$ docker build -t runnerly/tokendealer .
Sending build context to Docker daemon 148.5 MB
Step 1/6 : FROM python:3.6
  ---> 21289e3715bd
Step 2/6 : COPY . /app
  ---> 01cebcda7d1c
Removing intermediate container 36f0d93f5d78
Step 3/6 : RUN pip install -r /app/requirements.txt
  ---> Running in 90200690f834
Collecting pyjwt (from -r /app/requirements.txt (line 1))
[...]
Successfully built d2444a66978d
```

PIP 安装依赖项后，它又指向/app/目录并再次运行来安装项目。当 pip 命令指向一个目录时，它会查找并运行 setup.py。

在 TokenDealer 项目中，当安装应用时，会同时在系统中添加 runnerly-tokendealer 脚本。此处未使用 virtualenv，这是因为容器已提供隔离环境，所以使用 virtualenv 显得多余。所以，将 runnerly-tokendealer 脚本与 Python 可执行文件安装在一起，就可在命令行中直接使用它们。

这就是此处 CMD 指令直接指向 runnerly-tokendealer(即执行容器时运行的命令)的原因。

最后，EXPOSE 指令让容器在 TCP 的 5000 端口(即 Flask 应用监听的端口)监听传入的请求。

注意，公开端口后，还需要在运行时将其映射到本地端口，才能将其桥接到宿主机。

可通过-p 选项来设置桥接的端口。下例中，容器将 5000 端口桥接到宿主机本地的 5555 端口。

```
$ docker run -p 5555:5000 -t runnerly/tokendealer
```

为构建一个包含完整功能的镜像，需要完成的最后一件事是在启动 Flask 应用前运行 Web 服务器。

下一节分析该怎么做。

10.4　完整的栈——OpenResty、Circus 和 Flask

当使用 Docker 镜像发布微服务时，有两种策略来包含一个 Web 服务器。

第一种策略是不使用 Web 服务器，而直接公开 Flask 应用。然后在独立的 Docker 容器中运行 Web 服务器(如 OpenResty)，并将请求代理到 Flask 容器中。

然而，如果使用 nginx 中的一些增强特性(如第 7 章提到的基于 lua 的应用防火墙)，那么最好在同一容器中包含所有内容和一个专有的进程管理器。

在图 10-1 中，Docker 容器实现了第二种策略，它同时运行 Web 服务器和 Flask 服务，并使用 Circus 来启动和监视一个 nginx 进程和若干个 Flask 进程：

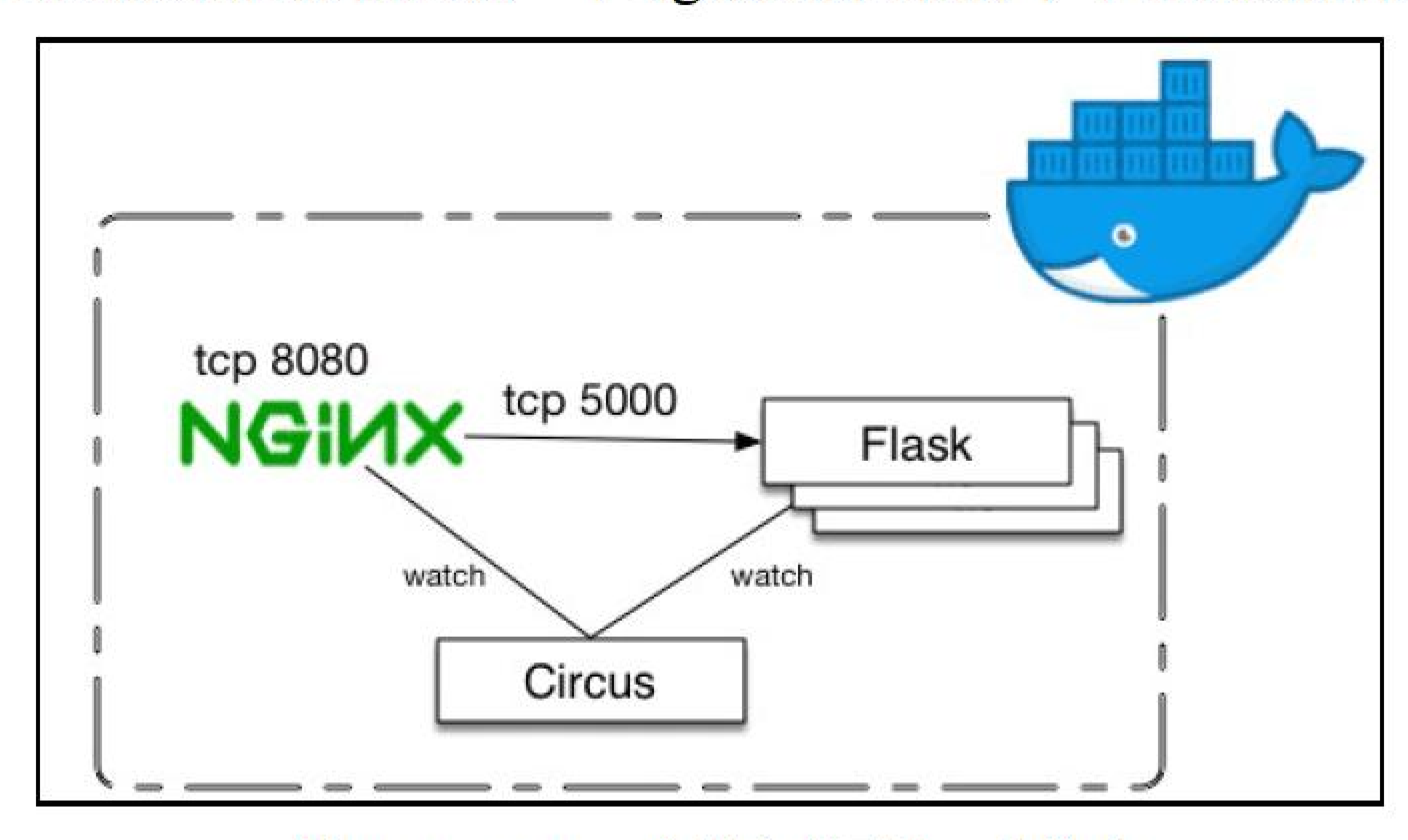


图 10-1　Docker 容器实现了第二种策略

本节通过以下步骤来完成这个容器：

(1) 下载、编译和安装 OpenResty；

(2) 添加一个 nginx 配置文件；

(3) 下载、安装 Circus 与 Chaussette；

(4) 添加 Circus 配置文件来运行 nginx 和 Flask 应用。

10.4.1 OpenResty

基础 Python 镜像可使用 Debian 的 apt 软件包管理器，但稳定版的 Debian 仓库并不包含 OpenResty(一个包含 lua 和其他扩展的 nginx 发行版)。不过，编译 OpenResty 的源码来编译并安装也很简单。

在 Dockerfile 中，首先确保 Debian 环境包含编译 OpenResty 需要的所有软件包。以下指令先更新软件包列表，然后安装所需的一切：

```
RUN apt-get -y update && \
   apt-get -y install libreadline-dev libncurses5-dev && \
   apt-get -y install libpcre3-dev libssl-dev perl make
```

注意以上代码将 3 条命令合并到一个 RUN 指令来减少 Dockerfile 中的指令数量。这样能减少最终镜像的大小。

下一步下载 OpenResty 的源码并编译。Python 基础镜像已包含 cURL，结合 tar 模块，可使用管道从 URL 解压 OpenResty 的压缩包：

以下的 configure 和 make 命令来自 OpenResty 文档，会编译和安装所有必需项：

```
RUN curl -sSL https://openresty.org/download/openresty-
1.11.2.3.tar.gz \
       | tar -xz && \
       cd openresty-1.11.2.3 && \
       ./configure -j2 && \
       make -j2 && \
       make install
```

编译完成后，会将 OpenResty 安装在/usr/local/openresty 中。可添加一个 ENV 指令，让容器的 PATH 变量包含 nginx 可执行文件：

```
ENV PATH "/usr/local/openresty/bin:/usr/local/openresty/nginx/
sbin:$PATH"
```

使用 OpenResty 前的最后一件事是包含一个 nginx 配置文件，并用它来启动 Web 服务器。

在下面这个简短示例中，nginx 将发送给 8080 端口的所有请求代理到 5000 端口(如图 10-1 所示)：

```
worker_processes 4;
error_log /logs/nginx-error.log;
daemon off;

events {
   worker_connections 1024;
}

http {
   server {
      listen 8080;

      location / {
         proxy_pass http://localhost:5000;
         proxy_set_header Host $host;
         proxy_set_header X-Real-IP $remote_addr;
      }
   }
}
```

注意 error_log 路径使用/logs/目录。这是日志在容器中的根目录。

需要通过 RUN 指令来创建该目录，并通过 VOLUME 指令来添加挂载点：

```
RUN mkdir /logs
VOLUME /logs
```

这样就能将/logs 目录挂载到宿主机的本地目录上。在运行时，如果容器被强制关闭，日志文件依然存在。



应该永远将 Docker 的文件系统视为易失性卷，它可能在任何时候丢失。

如果容器中的进程生成任何有用的东西，应该将结果数据复制到作为卷而挂载到容器之外的目录中。

这是一个完整的 nginx 配置文件，可通过 nginx 的-c 选项直接使用：

```
$ nginx -c nginx.conf
```

nginx 运行后，会假设 Circus 在 5000 端口监听传入的 TCP 连接，并且 nginx 自身

监听 8080 端口。

现在配置 Circus，将其绑定到一个套接字上，并衍生出若干 Flask 进程。

10.4.2 Circus

如果复用第 9 章的 Circus 和 Chaussette 配置，Circus 就能在 5000 端口绑定一个套接字，衍生出若干个 Flask 进程，然后在这个套接字上接收连接。Circus 还能监听在容器中运行的单个 nginx 进程。

为在容器中将 Circus 作为进程管理器来使用，首先安装 Circus 和 Chaussette，如下：

```
RUN pip install circus chaussette
```

下面的 Circus 配置与上一章中的配置类似(除了针对 nginx 的额外部分)：

```
[watcher:web]
cmd = runnerly-tokendealer --fd $(circus.sockets.web)
use_sockets = True
numprocesses = 5
copy_env = True

[socket:web]
host = 0.0.0.0
port = 5000

[watcher:nginx]cmd = nginx -c /app/nginx.confnumprocesses = 1copy_env
=True
```

这里使用 copy_env 标识，于是 Circus 以及衍生的进程都能访问容器的环境变量。由于 Dockerfile 文件设置了 PATH 变量，因此设置 copy_env 后，配置文件可直接调用 nginx 命令，而不需要指定路径。

创建这个 INI 文件后，就能使用 circusd 命令来加载它。

综合上述所有修改，最终 Dockerfile 文件如下所示：

```
FROM python:3.6

# OpenResty installation
RUN apt-get -y update && \
    apt-get -y install libreadline-dev libncurses5-dev && \
    apt-get -y install libpcre3-dev libssl-dev perl make
```

```
    RUN curl -sSL https://openresty.org/download/openresty-
1.11.2.3.tar.gz \
        | tar -xz && \
        cd openresty-1.11.2.3 && \
        ./configure -j2 && \
        make -j2 && \
        make install
    ENV PATH "/usr/local/openresty/bin:/usr/local/openresty/nginx/
sbin:$PATH"

    # config files
    COPY docker/circus.ini /app/circus.ini
    COPY docker/nginx.conf /app/nginx.conf
    COPY docker/settings.ini /app/settings.ini
    COPY docker/pubkey.pem /app/pubkey.pem
    COPY docker/privkey.pem /app/privkey.pem

    # copying the whole app directory
    COPY . /app

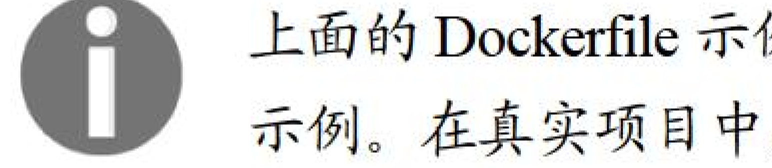
    # pip installs
    RUN pip install circus chaussette
    RUN pip install -r /app/requirements.txt
    RUN pip install /app/

    # logs directory
    RUN mkdir /logs
    VOLUME /logs

    # exposing Nginx's socket
    EXPOSE 8080

    # command that runs when the container is executed

    CMD circusd /app/circus.ini
```



上面的 Dockerfile 示例将 SSH 密钥拷贝到容器中以直接使用，但这只是简化示例。在真实项目中，应该将容器外部的密钥挂载到容器中。

假定 Dockerfile 文件位于微服务项目/docker 子目录中，可使用以下命令来构建和运行：

```
$ docker build -t runnerly/tokendealer -f docker/Dockerfile .
$ docker run --rm --v /tmp/logs:/logs -p 8080:8080 --name tokendealer -it
runnerly/tokendealer
```

在上例中，将/logs 挂载到本地/tmp/logs 目录中，于是日志就会被写入其中。

Docker 命令的-i 选项能确保在使用 Ctrl+C 停止容器时，将终止信号转发给 Circus 以便它正确关闭一切。在终端运行 Docker 容器时，这个选项很有用。如果不使用-i，那么用 Ctrl+C 强行关闭容器时，Docker 容器还在运行，然后需要使用 docker 终止命令来手动终止它。

--rm 选项会在容器停止时将其删除，--name 选项给 Docker 容器指定 Docker 环境的唯一名称。

这个 Dockerfile 示例还有很多地方可调整。例如，如果想从容器外部与 Circus 交互，可将用于控制 Circus 守护进程的 Circus UI(包括 Web 和命令行)所使用的套接字公开。

还可公开其他一些运行选项(例如需要启动的 Flask 进程数量)，可在运行时通过 Docker 的-e 选项将对应的环境变量传入。



可运行的完整 Dockerfile 文件见 https://github.com/Runnerly/tokendealer/tree/master/docker。

下一节将介绍容器之间如何交互。

10.5 基于 Docker 的部署

在容器运行微服务后，就需要让它们之间交互。由于我们将容器的套接字与宿主机上的本地套接字做了桥接，因此外部客户端能非常透明地使用容器。每个主机可有一个公开的 DNS 或 IP，程序可使用它连接到不同服务上。换句话说，只要主机 A 和主机 B 有公开的地址，并公开了与容器套接字进行桥接的本地套接字，那么位于主机 A 上的容器中的服务，就能与主机 B 上的容器中的服务交互。

不过，当两个容器运行在同一宿主机时——特别地，某个容器不需要对宿主机公开时——使用公共 DNS 来交互并非最佳方式。例如，如果 Docker 中的容器仅为内部所需(例如缓存服务)，那么应该将其限制为仅能从 localhost 访问。

为方便地实现这个场景，Docker提供了用户自定义网络功能，以便允许创建本地

虚拟网络，将容器添加到网络。如果使用--name选项运行容器，那么Docker会扮演DNS解析器角色，使用容器名就可在网络中访问它们。

下面使用 network 命令来创建一个名为 runnerly 的网络：

```
$ docker network create -driver=bridge runnerly
4a08e29d305b17f875a7d98053b77ea95503f620df580df03d83c6cd1011fb67
```

网络创建后，可使用--net 选项在这个网络中运行容器。下面使用 tokendealer 作为名称来运行容器：

```
$ docker run --rm --net=runnerly --name=tokendealer -v /tmp/logs:/logs -p
5555:8080 -it runnerly/tokendealer
2017-05-18 19:42:46 circus[5] [INFO] Starting master on pid 5
2017-05-18 19:42:46 circus[5] [INFO] sockets started
2017-05-18 19:42:46 circus[5] [INFO] Arbiter now waiting for commands
2017-05-18 19:42:46 circus[5] [INFO] nginx started
2017-05-18 19:42:46 circus[5] [INFO] web started
```

如果在同一网络中，用同样的镜像和不同名称来运行第二个容器，那么可在这个容器中使用容器名 tokendealer 直接连接第一个容器。

```
$ docker run --rm --net=runnerly --name=tokendealer2 -v /tmp/logs:/logs -p
8082:8080 -it runnerly/tokendealer ping tokendealer
PING tokendealer (172.20.0.2): 56 data bytes
64 bytes from 172.20.0.2: icmp_seq=0 ttl=64 time=0.474 ms
64 bytes from 172.20.0.2: icmp_seq=1 ttl=64 time=0.177 ms
64 bytes from 172.20.0.2: icmp_seq=2 ttl=64 time=0.218 ms
^C
```

在部署时，为微服务指定专有 Docker 网络是一个好做法——就算只有一个容器在运行也同样如此。你可随时在同一网络中添加新容器，或从 shell 调整网络权限。



Docker 还有其他网络策略，见 https://docs.docker.com/engine/userguide/networking/。

当部署多个容器才能运行一个微服务时，需要确保所有容器在启动时都被正确配置。

为简化配置，Docker 有一个称为 Docker Compose 的高级工具，将在下一节中介绍。

10.5.1 Docker Compose

在同一主机上运行多个容器时，为添加容器名称和网络以及绑定套接字，可能需要执行很长的命令。

通过在单个配置文件中定义多个容器的配置，Docker Compose(https://docs.docker.com/compose/)定义简化了上述运行多个容器的工作。

在 macOS 和 Windows 上安装 Docker 时，已经预先安装了这个实用程序。对 Linux 发行版来说，需要获取脚本并将其添加到系统中。它是单个脚本文件，可通过 PIP 来下载和安装(参见 https://docs.docker.com/compose/install/)。

将这个脚本安装到系统后，需要创建一个名为 docker-compose.yml 的 YAML 格式的文件，然后在其中的 services 罗列出所有 Docker 容器。



Compose 配置文件有很多选项，允许定义包含多个容器的部署的方方面面。它取代通常置于 Makefile 用来设置和运行容器的所有命令。这个 URL 列出所有选项：https://docs.docker.com/compose/compose-file/。

下例中，这个文件位于 Runnerly 的某个微服务中，它定义了两个服务：其中 microservice 会读取本地 Dockerfile，另一个 redis 使用来自 Docker Hub 的 Redis 镜像：

```
version: '2'
networks:
  runnerly:
services:
  microservice:
    networks:
      - runnerly
    build:
        context: .
        dockerfile: docker/Dockerfile
    ports:
      - "8080:8080"
    volumes:
      - /tmp/logs:/logs
  redis:
    image: "redis:alpine"
    networks:.
    - runnerly
```

Compose 文件还在 networks 部分创建网络，因此在部署容器前不必手动创建网络。

可使用 up 命令来构建和运行这两个容器：

```
$ docker-compose up
Starting tokendealer_microservice_1
Starting tokendealer_redis_1
Attaching to tokendealer_microservice_1, tokendealer_redis_1
[...]
redis_1         | 1:M 19 May 20:04:07.842 * DB loaded from disk: 0.000
seconds
redis_1         | 1:M 19 May 20:04:07.842 * The server is now ready to
accept connections on port 6379
microservice_1  | 2017-05-19 20:04:08 circus[5] [INFO] Starting master on
pid 5
microservice_1  | 2017-05-19 20:04:08 circus[5] [INFO] sockets started
microservice_1  | 2017-05-19 20:04:08 circus[5] [INFO] Arbiter now waiting
for commands
microservice_1  | 2017-05-19 20:04:08 circus[5] [INFO] nginx started
microservice_1  | 2017-05-19 20:04:08 circus[5] [INFO] web started
```

首次执行上述命令时会创建 microservice 镜像。

当你希望为微服务提供一个完整工作栈时，使用 Docker Compose 是非常适合的，它包含运行微服务的所有软件。

例如，要使用 Postgres 数据库，可通过 Postgres 镜像(https://hub.docker.com/_/postgres/)，在 Docker Compose 文件中将其链接到服务。

为演示软件或开发目的，最好容器化一切——甚至包括数据库。然而，如前所述，应该将 Docker 容器当作易失性文件系统。如果在容器中运行数据库，则要确保写入数据的目录被挂载到宿主机的文件系统上。

然而，大多数情况下，在生产环境下，数据库服务通常位于专有服务器上。因此使用容器来运行数据库没有多大意义，反而增加了一些开销和风险。

到目前为止，本章介绍如何在 Docker 中运行应用，如何在每个主机上部署多个容器，以及如何让它们交互。

当部署需要扩展的微服务时，通常要运行同一服务的多个实例以便支撑负载。

第 11 章讨论并行地运行同一容器的多个示例的多种选择。

10.5.2　集群和初始化简介

可通过运行分布在一个或多个主机上的多个容器来大规模部署微服务。

创建 Docker 镜像后，每个运行 Docker 守护进程的主机都可在物理资源受限的条件下运行任意数量的容器。当然，如果在同一主机上运行同一容器的多个实例，那么需要给每个实例使用不同的名称和套接字端口，以便区分它们。

运行同一镜像的一组容器被称为集群，已有一些用来管理集群的工具。

Docker有一个内置的集群特性称为swarm模式(https://docs.docker.com/engine/swarm/)。这个模式有一组令人印象深刻的特性，允许使用实用程序来管理所有集群。

部署集群后，需要搭建一个负载均衡器以使集群中的所有实例分担负载。例如，nginx 或 HAProxy 都可作为负载均衡器，作为入口，将传入的请求分发到集群中。

当 Docker 本身试图提供工具来处理容器的集群时，管理集群就显得很复杂。如果做法得当，就需要在宿主机之间共享一些配置，并确保启动和关闭容器是局部自动化的。例如，还需要“服务发现”特性来保证负载衡器能监测到容器的添加和移除。

诸如 Consul(https://www.consul.io/)或 Etcd(https://coreos.com/etcd/)的工具可用来发现服务和共享配置，将 Docker 配置为 swarm 模式就可与这些工具交互。

设置集群的另一个方面是初始化(Provision)。术语“初始化”指在给定所部署软件栈的描述后(这种描述是某种声明形式)，创建新主机和集群的过程。

例如，一个简单的初始化工具可以是遵循以下步骤的自定义 Python 脚本：

(1) 通过若干 Docker Compose 文件，读取描述了所需实例的配置文件。

(2) 在云厂商上启动若干个虚拟机。

(3) 等待所有虚拟机启动并运行。

(4) 确保在虚拟机中运行服务所需的一切都设置完毕。

(5) 在每个虚拟机上与 Docker 守护进程交互，并启动若干个容器。

(6) 连接任何需要连接(ping)的服务，确保新实例相互连通。

一旦能自动部署容器，如果一些虚拟机崩溃了，就可用自动化工具来分离它们。

不过，在 Python 脚本中完成上述所有工作有其局限性，但一些专用工具，如 Ansible(https://www.ansible.com/)或 Salt(https://docs.saltstack.com)，提供对 DevOps 友好的环境，可用来部署和管理宿主机。

Kubernetes(https://kubernetes.io/)是另一个工具，可用来在主机上部署集群。与 Ansible 或 Salt 不同，Kubernetes 擅长部署容器，并尝试提供一个能处处运行的通用方案。

例如，Kubernetes 可通过主流云厂商的 API 与它们交互，这意味着一旦定义了应用的部署方式，就可将其部署在 AWS、Digital Ocean、OpenStack 等云厂商上。然而，这引出一个问题——这是否对项目有用？

通常，如果已为应用选择云厂商，但出于某些原因决定迁移到另一云厂商上，那么这种情况很少像搭建一套新软件栈那么简单。有许多微妙细节让转换变得更复杂，

而且部署方式几乎各不相同。例如，一些云厂商提供数据存储方案，比自行搭建 PostgreSQL 或 MySQL 更便宜，而另一些云厂商的缓存服务比自行搭建 Redis 实例更昂贵。

一些团队跨多个云厂商来部署服务，但通常来说，他们不会将同样的微服务部署到多个提供商上。因为这样会让集群管理变得过于复杂。

此外，每个主流的云厂商都提供一系列完整的内置工具来管理托管的应用，包括负载均衡、可发现性和自动扩展。通常这些工具是部署微服务集群的最简单选项。

第 11 章将介绍如何使用 AWS 来部署应用。

总之，部署微服务的工具依赖于部署位置。如果自行管理服务器，那么 Kubernetes 可作为一个优秀方案来自动执行很多步骤，它可直接安装在 Linux 发行版(例如 Ubuntu)上。这个工具可基于 Docker 镜像来部署应用。

如果选择了托管方案，那么在投资工具集前，第一步要看一下云厂商已经提供了哪些工具。

10.6　本章小结

本章解释如何使用 Docker 来容器化微服务，以及如何基于 Docker 镜像来部署。

Docker 虽然还是年轻的技术，但对于生产环境已足够成熟。一定要记住最重要的事情：容器化应用可能在任何时间被摧毁，此时任何没有挂载到外部的数据都会丢失。

对于初始化并在集群中运行服务，现在还没有通用方案。有大量工具可供使用，可将它们组合成一个好方案。现在这个领域中有很多创新，最佳选择取决于服务部署在什么地方，以及团队如何工作。

处理这个问题的最佳方式是循序渐进地手动部署，然后在必要时自动化。自动化是好的，但如果你没有完全理解正在使用的工具集，它们可能很快变成噩梦。

这种情况下，云厂商为让其服务能更易于使用和有吸引力，在服务中内置了部署功能。其中，最大的玩家是 AWS(Amazon Web Services)。第 11 章将演示如何在 AWS 平台上部署微服务。当然，本书的目标并非让你使用 AWS——还有很多好方案——不过，可让你感受一下如何在托管服务上部署服务。

第 11 章

在 AWS 上部署

除非你和 Google 或 Amazon 一样需要运行成千上万的服务器，否则现在用数据中心管理硬件并没有太多好处。

云厂商提供一种比自行部署和维护基础设施更便宜的托管方案。Amazon Web Services(AWS)和其他海量云服务能让你用 Web 控制台管理虚拟机，而且这些云服务每年会增加很多新功能。

例如，AWS 新增的功能之一是 Amazon Lambda。当你的部署环境发生一些事情时，Lambda 可触发一个 Python 脚本。通过 Lambda，不必考虑设置服务器和配置 cron job(定时任务)，也不必考虑传递消息的格式。AWS 负责在 VM 中自动执行脚本，而你只需要为执行时间买单即可。

结合 Docker 提供的内容，这种功能真正改变了应用在云的部署方式，并提供相当大的灵活性。例如，为应对一个随后会回落的活动高峰，不必花费太多资金来搭建支持活动高峰的服务。你可部署能容纳大量请求的世界级基础设施，而且大多数情况下，它会比运行自己的硬件更便宜。

某些情况下，使用自己的数据中心可能更省钱，但增加了维护负担，而且让它做到与云服务一样可靠也是一个挑战。



虽然在媒体报道中云服务中断曾成为热点新闻，但 Amazon 或 Google 的中断服务极其罕见(一年中最多发生几小时)，它们有很高的可靠性。如 EC2服务级别协议(Service Level Agreement)保证每个地区最少99.95%的正常运行时间，否则你可拿回部分退款。实际上，正常运行时间通常高达99.999%。

可用在线工具 https://cloudharmony.com/status-1year-for-aws 来追踪云厂商的正常运行时间；不过，由于一些潜在的运行中断没有统计进来，结果不一定准确。

本章将完成以下两件事情：

- 探索 AWS 提供的一些功能；
- 在 AWS 上发布一个 Flask 应用。

本章的目标不是发布整个应用栈，因为这会花费很长时间，但会提供总览信息，来帮助你了解在 AWS 上如何部署微服务。

下面开始概述 AWS 的功能。

11.1 AWS 总览

亚马逊 Web 服务(Amazon Web Services，即 AWS)始于 2006 年，最早从 Amazon Elastic Compute Cloud(Amazon EC2)开始，并不断扩展。现在，AWS 已经有了无数服务。本章不讨论所有服务，只关注开始部署微服务时通常使用的服务，如图 11-1 所示。

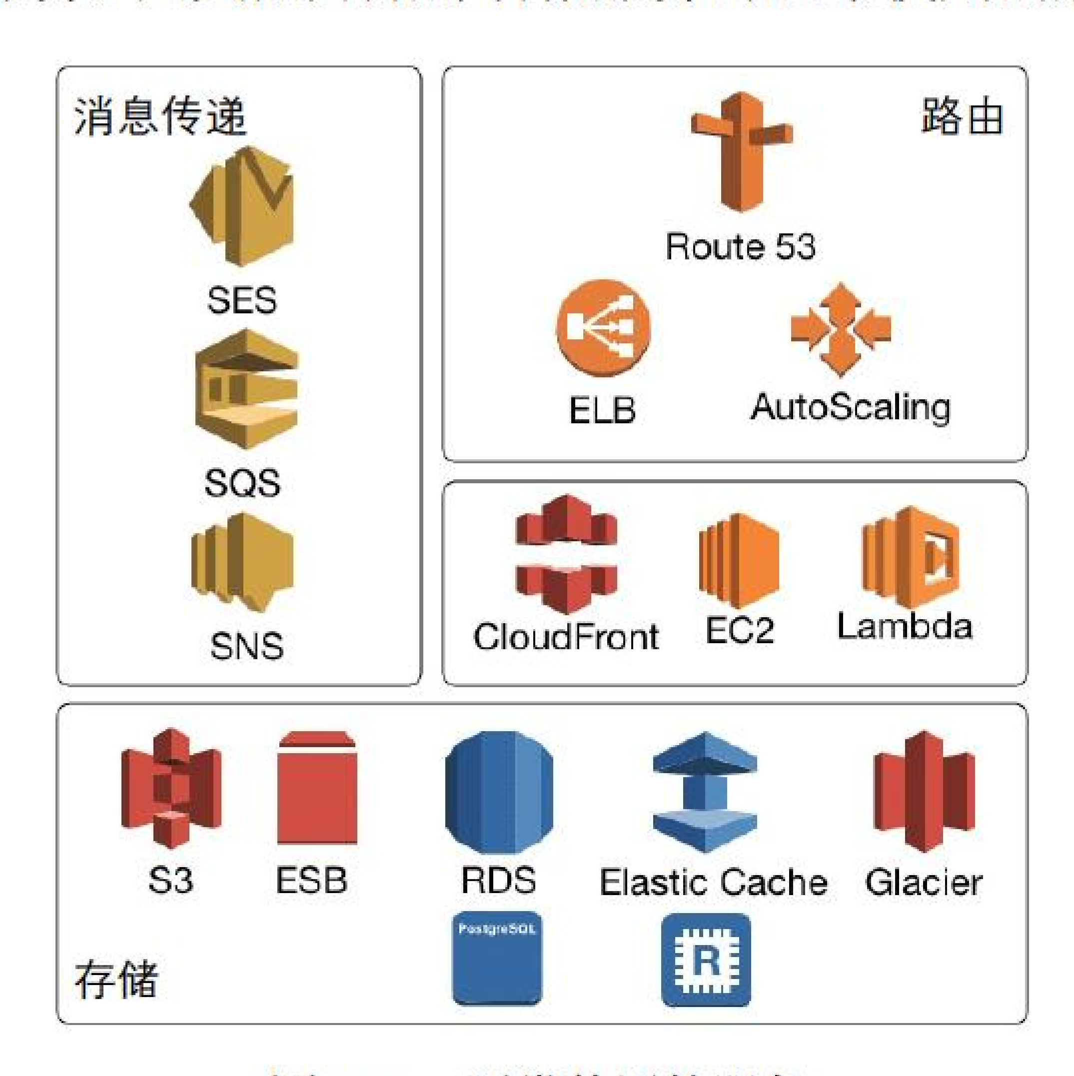


图 11-1　通常使用的服务

我们感兴趣的 AWS 服务可组织成图 11-1 所示的 4 个主要服务组:

- **路由**：用来把请求重定向到正确位置的服务。如 DNS 服务、负载均衡服务。
- **执行**：用来执行代码的服务，如 EC2 或 Lambda。
- **存储**：用来存储数据卷、缓存、常规数据库、长期贮存的服务或 CDN。
- **消息传递**：用来发送通知、邮件等的服务。

还有一个服务组未在图中显示，该服务负责与初始化资源和部署有关的一切。

下面具体介绍每个组。



如果想阅读 Amazon 服务的官方文档，常用地址是每个服务的根页面：https://aws.amazon.com/<*service name*>。

11.2　路由：Route53、ELB 和 AutoScaling

Route53(https://aws.amazon.com/route53/)指用于 DNS 服务器的 TCP 端口 53，这是 Amazon 的 DNS 服务。如在 BIND(http://www.isc.org/downloads/bind/)所做的一样，可在 Route53 中定义 DNS 条目，通过设置这个服务将请求自动路由到特定 AWS 服务上，这些服务可能承载应用或文件。

DNS 是部署的关键部分。它需要高度可用，并能尽快路由请求。如果在 AWS 上部署服务，强烈推荐使用 Route53，或联系出售域名的公司的 DNS 提供商，最好不要自行处理 DNS。

Route53 可与 ELB(Elastic Load Balancing，弹性负载均衡 https://aws.amazon.com/elasticloadbalancing/)密切协作，ELB 作为一个负载均衡器，可用来将传入的请求分发到多个后端。通常，如果要为同一微服务部署包含多个 VM 的群集，ELB 可用于分配负载。ELB 通过健康检查来监控所有实例，自动将不健康的节点从轮询中移除。

最后一个有趣的路由服务是 AutoScaling(https://aws.amazon.com/autoscaling/)。该服务基于某些事件自动添加实例。例如，如果一个节点无响应或崩溃，AutoScaling 会接收一个 ELB Health Check 事件。然后自动终止存在问题的 VM，并启动一个新的 VM。

通过这三个服务，可为微服务设置强健的路由系统。下一节将分析哪些服务可用于运行实际代码。

11.3　执行：EC2 和 Lambda

AWS 的核心是 EC2(https://aws.amazon.com/ec2/)，可用它创建虚拟机。Amazon 使用 Xen hypervisor(https://www.xenproject.org/)来运行虚拟机，使用 AMI (Amazon Machine Images)来执行安装。

AWS 有一个庞大的 AMI 列表可供选择；你也可调整一个现有 AMI 来创建自己的 AMI。AMI 和 Docker 镜像和用法类似。一旦从 Amazon 控制台选择 AMI，就可启动一个实例。启动后，可通过 SSH 进入并开始工作。

任何时候都可为 VM 创建快照并创建一个 AMI 来保存实例状态。这个功能的有用之处在于，要手动设置服务器，可将其用作部署集群的基础。

EC2 实例包含不同系列(https://aws.amazon.com/ec2/instance-types/)。T2、M3 和 M4 系列可用于通用目的。T 系列使用突增技术，当工作负载达到峰值时，会增强实例的基线性能。

C3 和 C4 系列适用于 CPU 密集型应用(最多可支持 32 个 Xeon CPU)，X1 和 R4 系列可提供大量内存(最多可支持 1952GB)。

当然，更多内存和 CPU 意味着实例更昂贵。对于 Python 微服务，假设没有在应用实例上承载任何数据库，那么 t2.xxx 或 m3.xxx 是很好的选择。但要注意避免使用 t2.nano 和 t2.micro，它们可用来运行测试，但在生产环境上运行任何东西都会受到限制。你需要选择的服务器大小取决于操作系统和应用占用的资源。

因为使用 Docker 镜像来部署微服务，所以不需要运行一个花哨的 Linux 发行包。唯一需要关注的是选择一个能运行 Docker 容器的 AMI。

在 AWS 中，内建的部署 Docker 容器方式是使用 EC2 Container Service(ECS)(https://aws.amazon.com/ecs)。ECS 提供的功能与 Kubernetes 相似，可很好地集成其他 AWS 服务。ECS 使用自己的 Linux AMI 来运行 Docker 容器，但你也可配置 ECS 来运行其他 AMI。例如，CoreOS(https://coreos.com/)是一个主要用来运行 Docker 容器的 Linux 版本。如果使用 CoreOS，就不会被锁定在 AWS 中。

最后，Lambda(https://aws.amazon.com/lambda/)是一个可用来触发 Lambda 函数的服务。Lambda 函数是可用 Node.js、Java、C#、Python 2.7(或 Python 3.6)编写的一段代码，可通过部署包(一个包含脚本和所有依赖项的 ZIP 文件)来部署。如果使用 Python，ZIP 文件通常是包含运行该函数所有依赖项的 virtualenv。

Lambda 函数可替代 Celery 职程，因为可通过 AWS 事件异步地触发它们。使用 Lambda 的好处在于不需要部署一个 7×24 小时运行的 Celery 微服务(这个微服务会不间断地从队列中捡起消息)。基于消息频率，使用 Lambda 可能降低成本。但重申一次，使用 Lambda 意味着你将被锁定在 AWS 服务中。

下面分析存储解决方案。

11.4 存储：EBS、S3、RDS、ElasticCache 和 CloudFront

创建 EC2 实例时，它会与一个或多个 EBS(Elastic Block Stores，https://aws.amazon.com/ebs/)一同工作。EBS 是一个复制的存储卷，EC2 实例挂载后将其用作文件系统。创建一个新 EC2 实例时，可创建一个新的 EBS，然后决定在 SSD(固态硬盘)还是 HDD(传统硬盘)上运行它，并设置它的初始化大小和其他一些参数。存储卷的费用

取决于你的选择。

S3(Simple Storage Service，https://aws.amazon.com/s3/)是一个存储服务，它使用存储桶(bucket)来组织数据。简单来说，存储桶是用来组织数据的名称空间。可将存储桶视为键-值(key-value)存储，值是要存储的数据。存储的数据没有上限，S3 为大文件在存储桶的进出提供所需的一切。S3 通常用于分发文件，因为每个存储桶的入口是唯一的公开 URL。通过配置 CloudFront 可将 S3 用作后端。

S3 提供的一个有趣特性是它可根据文件的读写频率提供不同的存储后端。例如，当你想存储大文件而且很少访问时，可用 Glacier(https://aws.amazon.com/glacier/)作为后端。一个使用场景是数据备份。从 Python 应用访问 S3 非常容易，在微服务中使用 S3 作为数据后端也很常见。

ElasticCache(https://aws.amazon.com/elasticache/)是一个缓存服务，提供两种后端——Redis 和 Memcached。ElasticCache 利用 Redis 的分片和复制功能，并允许部署 Redis 节点的集群。如果将大量数据保存在 Redis 中，可能耗尽内存容量，Redis 的分片机制可将数据分散在多个节点上来提高 Redis 容量。

RDS(Relational Database Service，https://aws.amazon.com/rds/)是一个数据库服务，可使用多种数据库作为后端，尤其是能使用 MySQL 和 PostgreSQL。



AWS 有一个在线计算器(http://calculator.s3.amazonaws.com/index.html)用来估计部署方案的成本。

使用 RDS 而不是自行部署数据库的最大好处在于，AWS 能管理节点的集群，这样不必完成任何维护工作，就能让数据库提供高可用性和可靠性。AWS 最近在 RDS 后端添加了 PostgreSQL，这让 RDS 变得非常受欢迎，这也是人们选择在 AWS 上部署应用的原因之一。

最近添加的另一个后端是 AWS 专有的 Amazon Aurora(https://aws.amazon.com/rds/aurora/details/)，它与 AWS 绑定，实现了 MySQL 5.x，但比普通 MySQL 速度更快(Amazon 声称速度能提升 5 倍)。

最后，CloudFront(https://aws.amazon.com/cloudfront/)是 Amazon 的 Content Delivery Network(CDN，内容分发网络)。当用户遍布全球时，使用 CDN 是处理静态文件的最佳方法。Amazon 会缓存文件，并将客户端的请求路由到最近的服务器，尽量缩短请求的延迟。CDN 可用来存放视频、CSS 和 JS 文件——唯一需要注意的是成本。如果微服务只需要提供少量静态文件，更简单的方式是直接用 EC2 实例提供。

11.4.1 消息：SES、SQS 和 SNS

对于所有消息需求，AWS 提供如下三个主要服务：

- 简单邮件服务(SES)：一个 Email 服务。
- 简单队列服务(SQS)：一个与 RabbitMQ 类似的队列。
- 简单通知服务(SNS)：一个发布/订阅系统和推送通知系统。

1. 简单邮件服务

如果服务要给用户发送邮件，难以确保是否最终成功发送到用户的收件箱中。如果从应用服务器使用本地 SMTP 服务来发送邮件，只有完成很多工作并正确配置系统，才能避免邮件被接收服务器标记成垃圾邮件。

此外，即使配置正确，但若发送服务器的 IP 在接受服务器的黑名单的 IP 组中(因为某个垃圾邮件发送者使用一个与你的 IP 接近的地址发送垃圾邮件)，那么除了试图从黑名单中删除 IP 外，你毫无办法。最糟的场景是使用垃圾邮件发送者用过的 IP。

由于确保邮件到达预期目的地是困难的，所以使用专门的第三方服务来发送邮件是个好主意——即便微服务不是部署在云上，也同样如此。

市场上有很多这样的第三方服务，AWS 有简单邮件服务(SES，https://aws.amazon.com/ses/)。用 SES 发邮件只需要使用 SES 的 SMTP 接口即可。它也提供一个 API，不过最好使用 SMTP，因为当执行一些开发或测试时，服务可使用本地 SMTP。

2. 简单队列服务

SQS(https://aws.amazon.com/sqs/)功能是 RabbitMQ 的子集，对大多数使用场景来说已经足够了。

可创建两类队列。先入先出(First-In-First-Out，FIFO)能按接收消息的顺序存储消息，并确保从队列中检索到的消息只能读取一次。当希望保存由职程获取的消息流时，它们非常有用，就像使用 Celery 和 Redis 所做的一样。这个队列的待处理消息上限是 20 000 条。

第二种类型(标准类型)与 FIFO 相似，但不能保证消息的顺序。它比 FIFO 队列更快，待处理消息的上限也更高(120 000)。

存储在 SQS 的消息会被复制到 AWS 云中的多个 AZ 上来实现可靠性。

AWS 按区域(Region)和区域中的可用区(Availability Zone，AZ)来组织。

区域彼此独立，以确保容错和稳定性。AZ 是低延迟的独立连接。可在 AWS 中的同一负载平衡器后使用多个 AZ 实例，以便在同一区域中跨不同的 AZ 实例。

因为一条消息的上限是 256KB，所以可在 FIFO 队列中存储 5GB 数据，在标准队列存储 30GB 数据。换句话说，除金钱外没有真正的限制。

3. 简单通知服务

消息工具中的最后一个服务是 SNS(https://aws.amazon.com/sns/)，它提供两个消息 API。

第一个是发布/订阅 API，可用来在应用中触发动作。发布者可以是 Amazon 服务或你的应用，订阅者可以是一个 SQS 队列、一个 Lambda 函数或任何 HTTP 端点，如微服务。

第二个是推送 API，可用来向移动设备发送消息。这种情况下，SNS 通过与第三方 API(如 Google Cloud Messaging，GCM)进行交互来抵达手机，或通过短信(SMS)发送文本消息。

SQS 和 SNS 服务可组合在一起，以取代自定义部署的消息系统，如 RabbitMQ。但需要检查它们的功能是否满足你的需要。

下一节将介绍可用于初始化资源和部署的 AWS 服务。

11.4.2　初始化资源和部署：CloudFormation 和 ECS

第 10 章讲过，有多种在云上初始化资源和部署 Docker 容器的方法。诸如 Kubernetes 工具可用来在 AWS 上管理所有运行的实例。

AWS 也提供自己的服务来部署容器化应用的集群；它被称为 EC2 Container Service-ECS(https://aws.amazon.com/ecs)，使用一个称为 CloudFormation(https://aws.amazon.com/cloudformation/)的服务来管理其他服务。

CloudFormation 允许通过 JSON 文件描述要在 Amazon 上运行的不同实例，并在 AWS 上自动操作一切，从部署实例到自动扩展。

ECS 基本是一个仪表盘集合，通过使用预定义模板，可实现集群的可视化，并操作通过 CloudFormation 部署的集群。运行 Docker 守护进程的 AMI 也是为这个目的所做的调整，如 CoreOS。

使用 ECS 的方便之处在于，只需要填写几张表单，就可在几分钟内为给定 Docker 镜像创建和运行集群。ECS 控制台为群集提供一些基本度量标准，并提供很多功能，如基于 CPU 或内存使用情况来规划一次新部署。

除了基于表单的最初设置外，由 ECS 部署的集群由 Task definition 驱动。Task definition 定义实例的完整生命周期，描述 Docker 镜像如何运行以及一些事件发生时的行为。

11.5　在 AWS 上部署简介

前面介绍了主要的 AWS 服务，下面尝试部署微服务。

要理解 AWS 如何工作，最好了解如何手工部署 EC2 实例，然后在其上运行微服务。本节描述如何部署 CoreOS 实例，在其中运行 Docker 容器，然后介绍使用 ECS 部署自动化集群，最后讨论如何使用 Route53 发布服务的集群，这些集群位于同一域名下。

首先创建 AWS 账户。

11.5.1　创建 AWS 账号

在 Amazon 上部署的第一步是在 https://aws.amazon.com 创建一个账户。需要输入信用卡信息来完成注册，但某些情况下，可使用基本计划(basic plan)免费使用一些服务。

免费提供的服务已经足以评估 AWS 了。

一旦完成注册，就会被重定向到 AWS 控制台。第一件事是从右上角的登录名下的菜单中选择 US East(N. Virginia)地区。北弗吉尼亚是用来设置特定计费警告的区域。

第二步在 Billing 控制台中配置警告，页面在 https://console.aws.amazon.com/billing/home#/，或从菜单导航到这里。在 Preferences 中选中 Receive Billing Alerts 复选框，如图 11-2 所示。

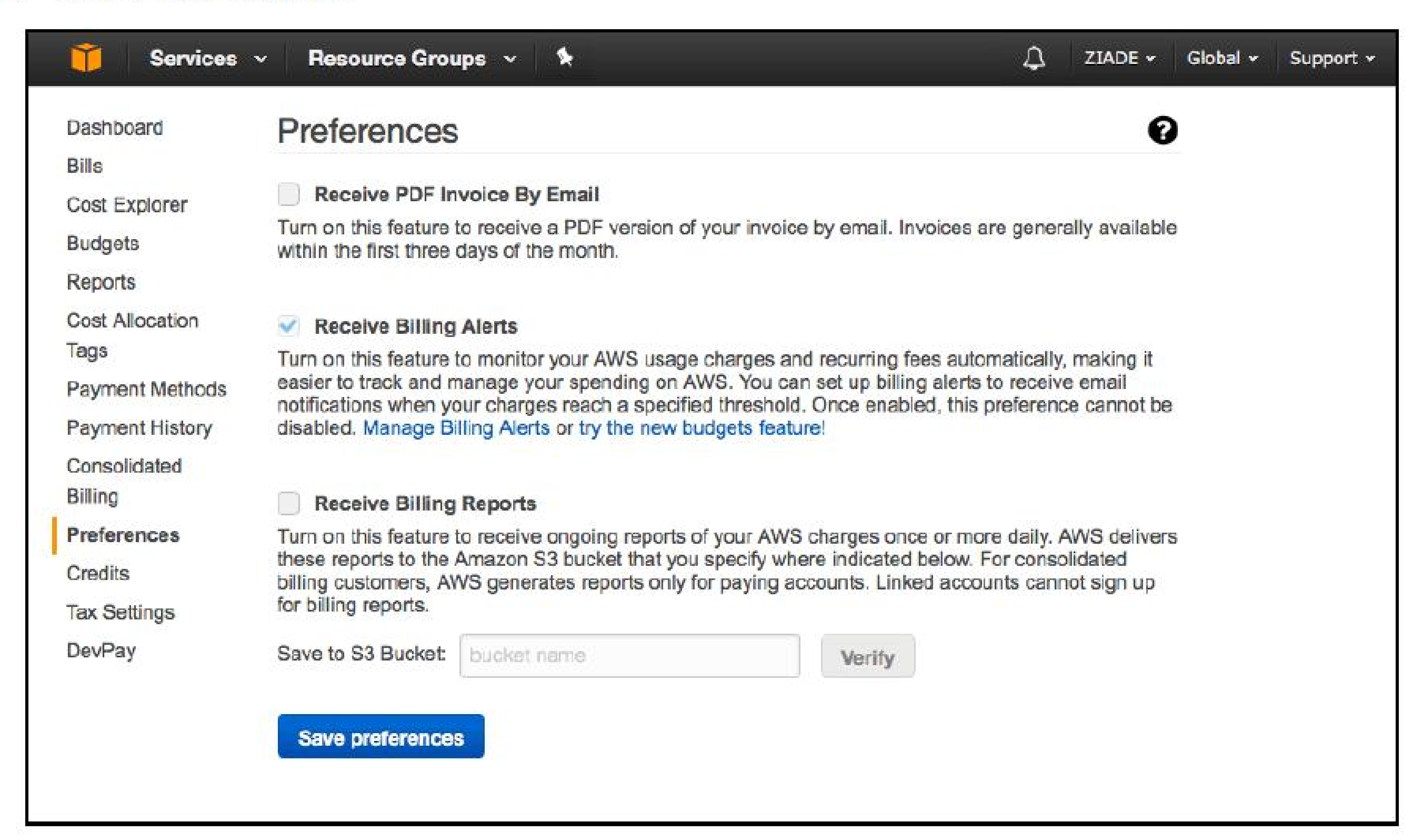


图 11-2　选中 Receive Billing Alerts 复选框

一旦完成，打开 CloudWatch(https://console.aws.amazon.com/cloudwatch/home)面板，选择左侧的 Alarms | Billing 创建一个新警告。此时弹出一个新窗口，可设置为当某个

服务开始花钱时发出提醒。此处可设置 0.01 美元作为最高收费限制。如果只进行测试，此通知可防止浪费钱，如图 11-3 所示。

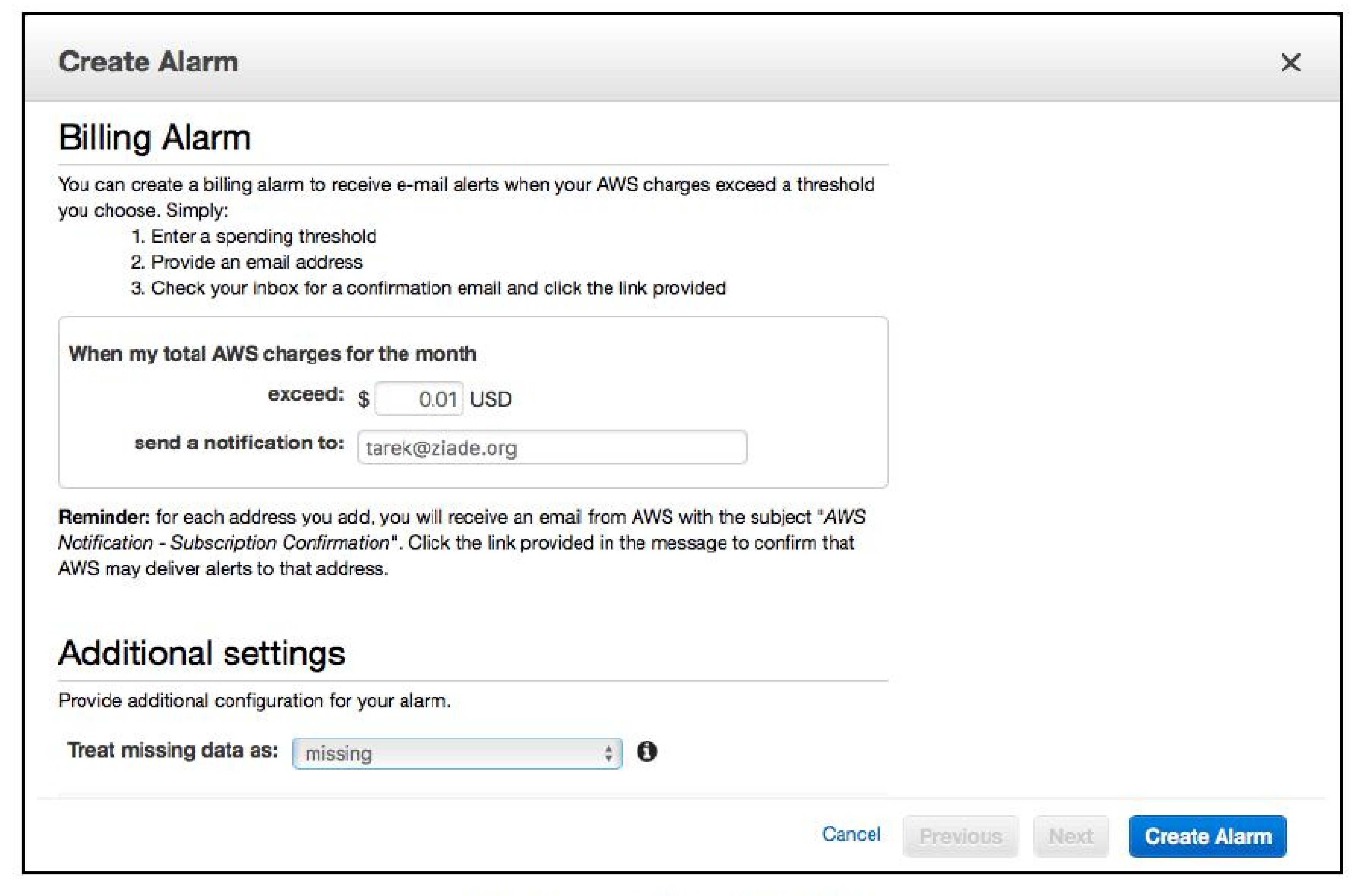


图 11-3　创建一个新警告

任何时候都可通过单击左上角的 Services 菜单来访问服务。它会打开一个包含所有服务的面板。

如果单击 EC2，会重定向到位于 https://console.aws.amazon.com/ec2/v2/home 的 EC2 控制台，在那里可创建一个新实例，如图 11-4 所示。

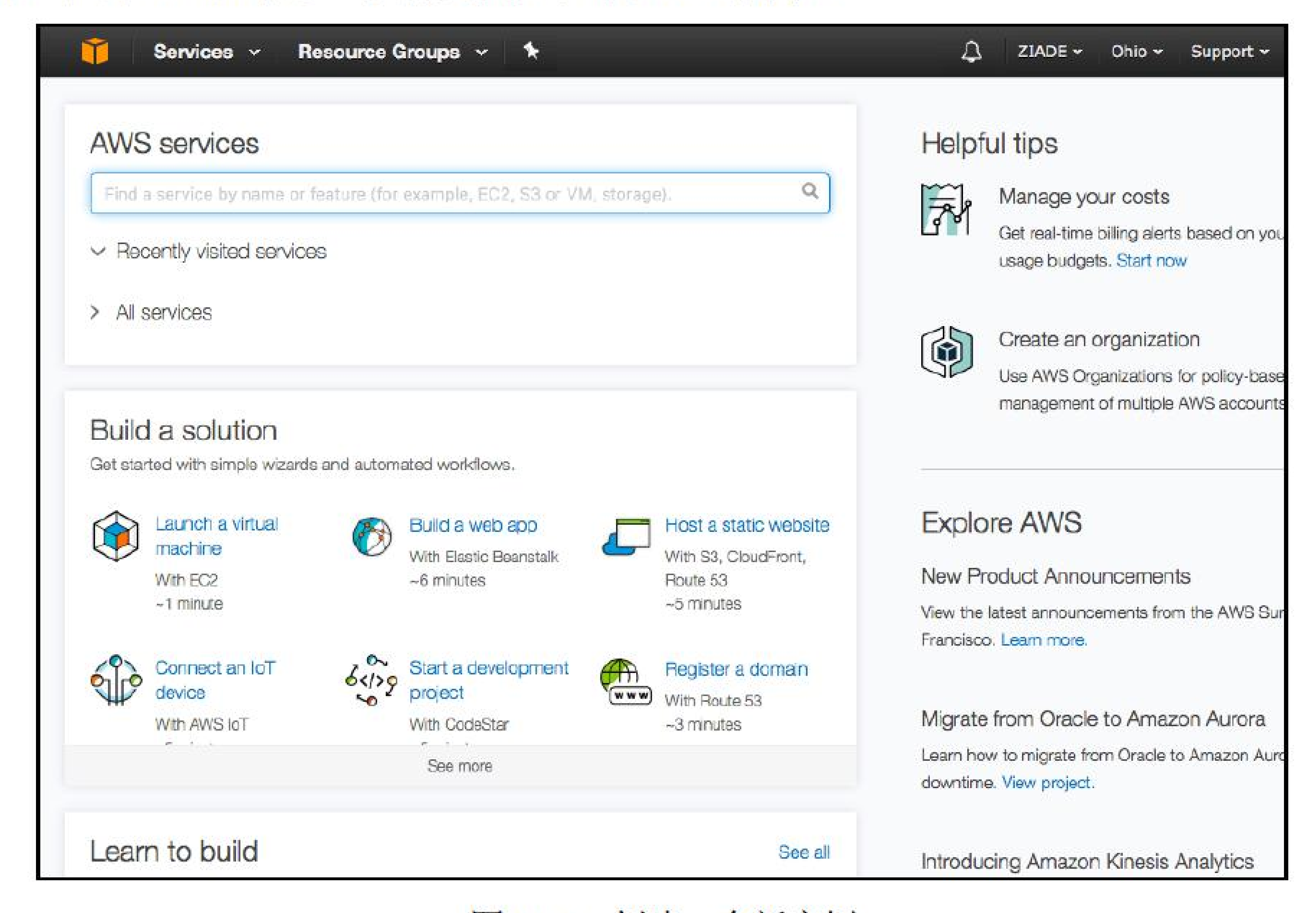


图 11-4　创建一个新实例

11.5.2 使用 CoreOS 在 EC2 上部署

单击 Launch Instance 蓝色按钮，选择一个 AMI 来运行新的 VM，如图 11-5 所示。

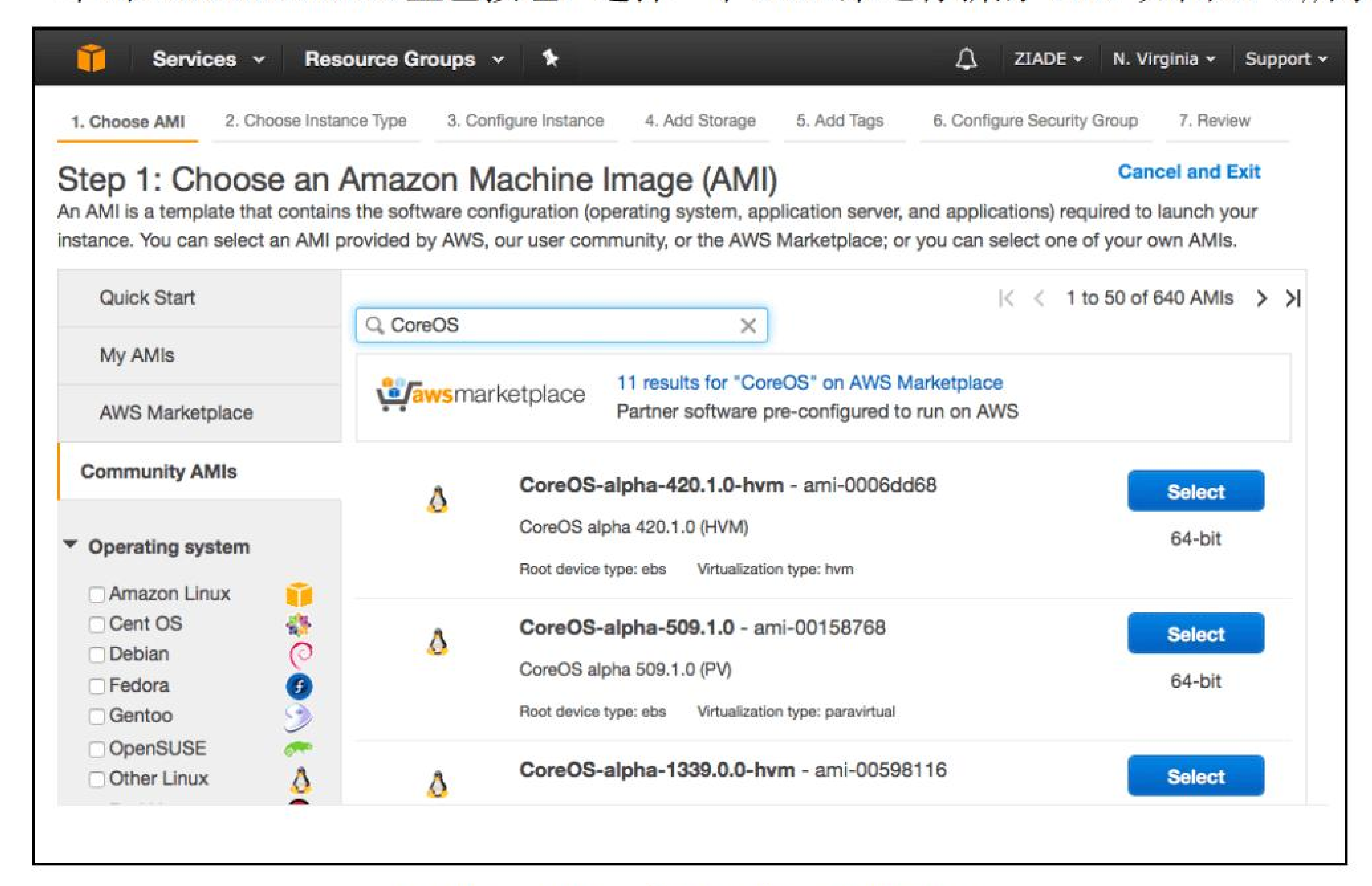


图 11-5 选择一个 AMI 来运行新的 VM

在 Community AMIs 下搜索 CoreOS 实例，会列出所有可用的 CoreOS AMI。

这里有两种 AMI：Paravirtual(PV)和 Hardware Virtual Machine(HVM)。这些是 Xen 管理程序中的两个虚拟化级别。PV 是完全虚拟化，HVM 是部分虚拟化。根据 Linux 不同的发行版，某些类型的 VM 可能无法在 PV 下运行。

如果只想尝试一下，在列表中选择第一个 PV AMI。然后在下一个页面选择 t1.micro，选择 Review And Launch 选项后继续。最后单击 Launch 按钮。

在创建 VM 前，控制台要求你创建一个新的 SSH 密钥对。如果想访问这个 VM，这是关键一步。应该给每个 VM 生成新密钥对，并给密钥对指定唯一名称，然后下载文件。将获得一个.pem 文件，可将它添加到~/.ssh 目录中，如图 11-6 所示。



提示：不要丢失此文件，出于安全考虑，Amazon 不会存储它。

一旦实例运行起来，就会显示在 EC2 控制台的列表中(可单击左侧的 Instances 菜单来查看)。

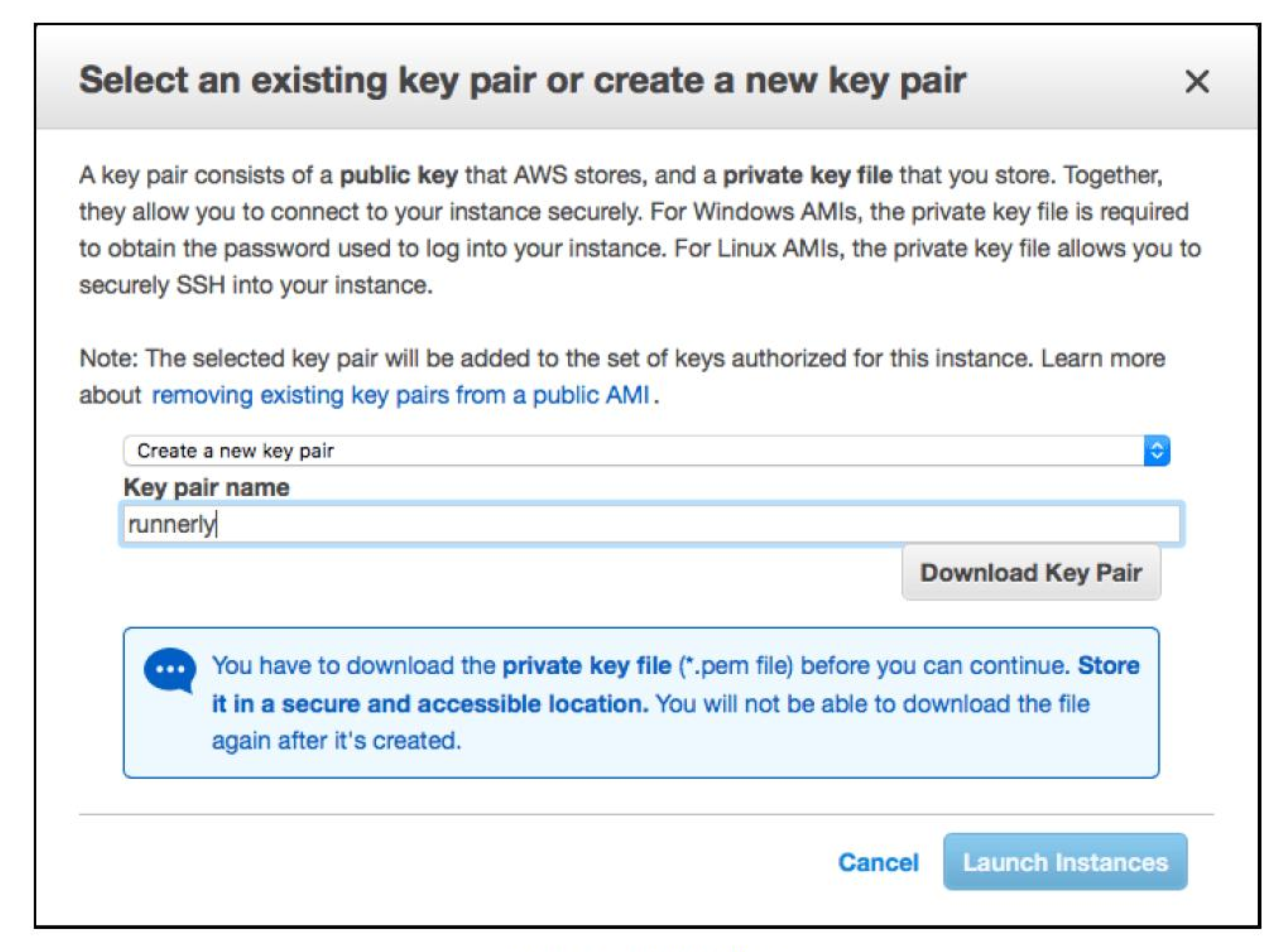


图 11-6　处理密钥对

可在 status checks 列中查看 VM 状态。AWS 部署 VM 会花费一些时间，一旦就绪，就能用 SSH 来操作 VM。这需要使用作为密钥的.pem 和 VM 的公开 DNS 地址。

CoreOS 的默认用户是 core，一旦连接，说明运行 Docker 容器需要的一切已经就绪。但需要更新它。当 CoreOS 持续更新时，可用 update_engine_client 命令强制系统进行更新，然后使用 sudo reboot 重启 VM。如下面的脚本所示：

```
$ ssh -i ~/.ssh/runnerly.pem
core@ec2-34-224-101-250.compute-1.amazonaws.com
CoreOS (alpha)
core@ip-172-31-24-180 ~ $
core@ip-172-31-24-180 ~ $ update_engine_client -update
[0530/083245:INFO:update_engine_client.cc(245)] Initiating update
check and
install.
[0530/083245:INFO:update_engine_client.cc(250)] Waiting for update to
complete.
LAST_CHECKED_TIME=1496132682
PROGRESS=0.000000
CURRENT_OP=UPDATE_STATUS_UPDATED_NEED_REBOOT
```

```
NEW_VERSION=0.0.0.0
NEW_SIZE=282041956
core@ip-172-31-24-180 ~ $ sudo reboot
Connection to ec2-34-224-101-250.compute-1.amazonaws.com closed by
remote
host.
Connection to ec2-34-224-101-250.compute-1.amazonaws.com closed.
```

VM 启动后，你将得到最新版本的 Docker。然后尝试从一个 busybox Docker 容器回显 hello，如下所示：

```
$ ssh -i ~/.ssh/runnerly.pem
core@ec2-34-224-101-250.compute-1.amazonaws.com
Last login: Tue May 30 08:24:26 UTC 2017 from 91.161.42.131 on pts/0
Container Linux by CoreOS alpha (1423.0.0)
core@ip-172-31-24-180 ~ $
docker -v Docker version 17.05.0-ce, build 89658be
core@ip-172-31-24-180 ~ $ docker run busybox /bin/echo hello
Unable to find image 'busybox:latest' locally
latest: Pulling from library/busybox
1cae461a1479: Pull complete
Digest:
sha256:c79345819a6882c31b41bc771d9a94fc52872fa651b36771fbe0c8461d7ee558
Status: Downloaded newer image for busybox:latest hello
core@ip-172-31-24-180 ~ $
```

如果前面的调用成功了，那么完整 Docker 环境就可以工作了。下面使用 Docker Hub 的 docker-flask 镜像运行 Web 应用：

```
core@ip-172-31-24-180 ~ $
docker run -d -p 80:80 p0bailey/docker-flask
Unable to find image 'p0bailey/docker-flask:latest' locally
latest:
Pulling from p0bailey/docker-flask
bf5d46315322: Pull complete
9f13e0ac480c: Pull complete
e8988b5b3097: Pull complete
40af181810e7: Pull complete
e6f7c7e5c03e: Pull complete
ef4a9c1b628c: Pull complete
```

```
d4792c0323df: Pull complete
6ed446a13dca: Pull complete
886152aa6422: Pull complete
b0613c27c0ab: Pull complete
Digest:
sha256:1daed864d5814b602092b44958d7ee6aa9f915c6ce5f4d662d7305e46846353b
Status: Downloaded newer image for p0bailey/docker-flask:latest
345632b94f02527c972672ad42147443f8d905d5f9cd735c48c35effd978e971
```

默认情况下，AWS 只为 SSH 访问打开端口 22。为访问端口 80，需要打开 EC2 控制台的 Instances 列表，单击为实例创建的安全组(名称通常为 launch-wizard-xx)，如图 11-7 所示。

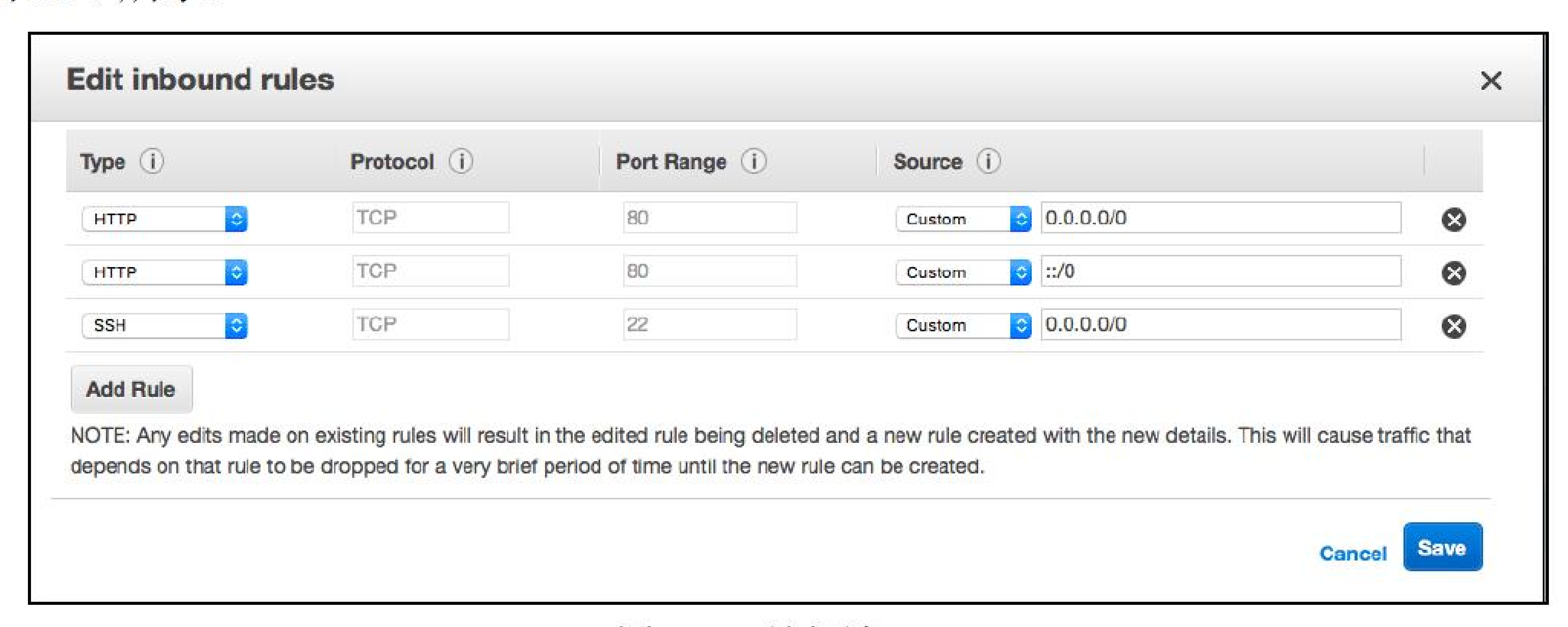


图 11-7　访问端口 80

单击它会弹出安全组页面，可在这里编辑 inbound rules(传入规则)来添加 HTTP。会立刻打开端口 80，然后可在浏览器上使用公共 DNS 来访问 Flask 应用。

这就是在 AWS 上运行 Docker 镜像需要完成的工作，是任何部署的基础。以此为基础，可创建一组实例来部署集群，这些实例由 AutoScaling 和 ELB 服务进行管理。

高级工具 CloudFormation 可通过定义模板来自动执行这些步骤。但在使用 Docker 时，ECS 是在 AWS 实现自动部署的终极杀器。下一节将介绍如何使用它。

11.6　使用 ECS 部署

前面介绍过，ECS 负责自动部署 Docker 镜像，并设置实例需要的其他服务。

这种场景下，不需要自行创建 EC2 实例。ECS 使用自己的 AMI，这个 AMI 专门用来在 EC2 上运行 Docker 容器。它很像 CoreOS，有一个 Docker 守护进程，但为了共享配置以及触发事件而与 AWS 基础设施进行集成。

ECS 集群部署由多个元素组成：

- Elastic Load Balancer(在 EC2 中)：用来给实例分发请求。
- Task definition：用来确定哪个 Docker 镜像需要部署，容器和主机之间应该绑定哪个端口。
- Service：使用 Task definition 来驱动 EC2 实例的创建，并在其中运行 Docker 容器。
- Cluster：组合多个 Service、多个 Task definition 和 ELB。

刚开始会觉得在 ECS 上部署群集很复杂，因为需要按特定顺序创建元素。例如，首先需要设置 ELB。

幸运的是，通过运行向导，将按正确顺序创建。首次在控制台上访问 ECS 服务时，将显示向导页面。打开登录页后，可单击 Get Started 按钮运行向导。

选中 Deploy a sample application onto an Amazon ECS Cluster 选项，然后继续，如图 11-8 所示。

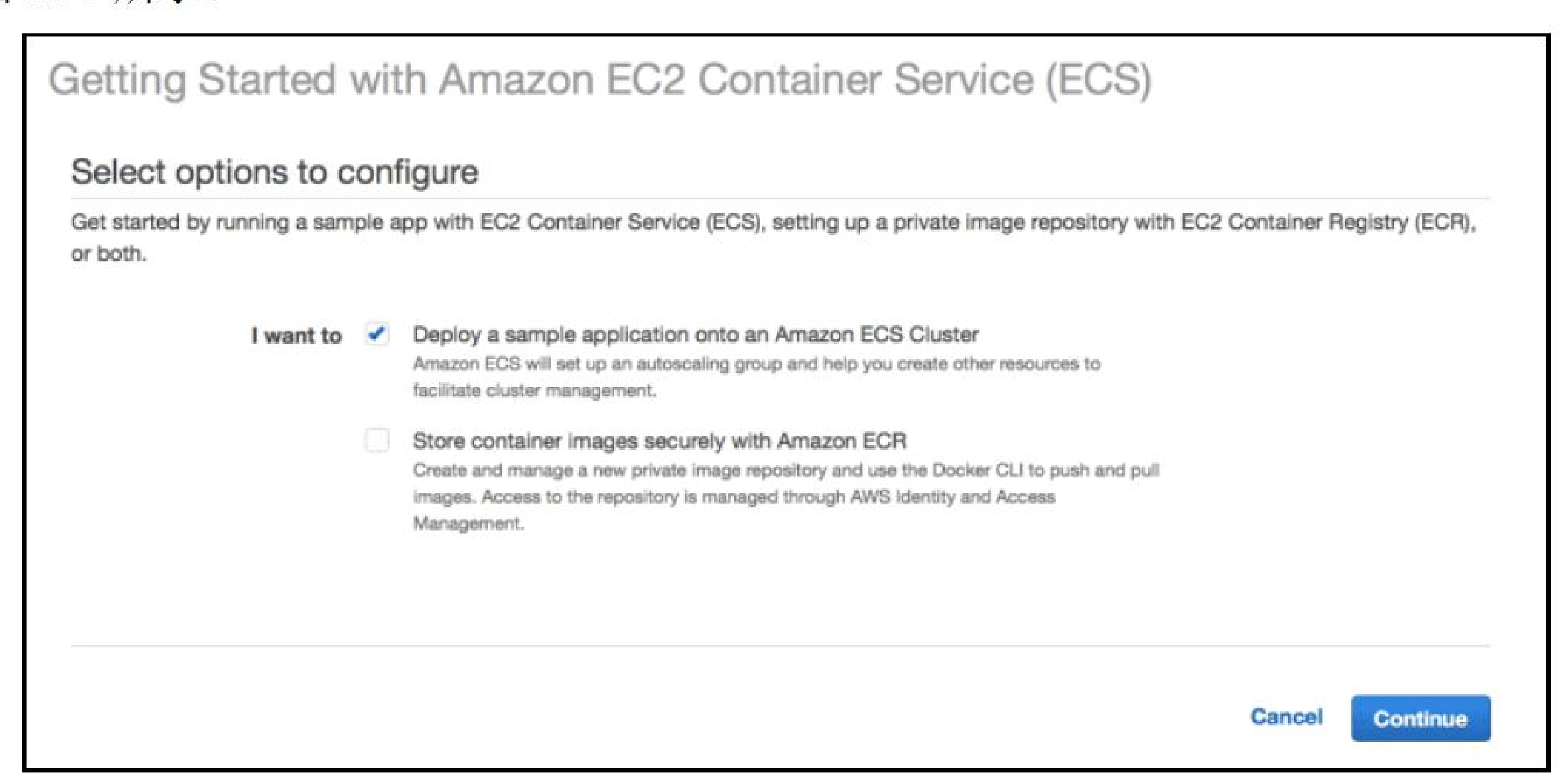


图 11-8　选中 Deploy a sample application onto an Amazon ECS Cluster 选项

此操作将显示 Create a task definition 对话框，可在其中定义任务名和运行该任务的容器，如图 11-9 所示。

上例部署了相同的 Flask 应用，与之前部署在 EC2 上的应用相同，所以可在 Task definition 的 Container 表单上提供镜像名。Docker 镜像需要位于 Docker Hub 或 AWS 自己的 Docker 镜像仓库中。

在这个表单中，也可设置所有 Docker 容器和主机系统的端口映射关系。镜像运行时会使用该选项。这里绑定端口 80，这是我们使用的 Docker 镜像公开的端口，Docker 中包含 Flask 应用。

向导的下一步是 Configure service 窗口，如图 11-10 所示。

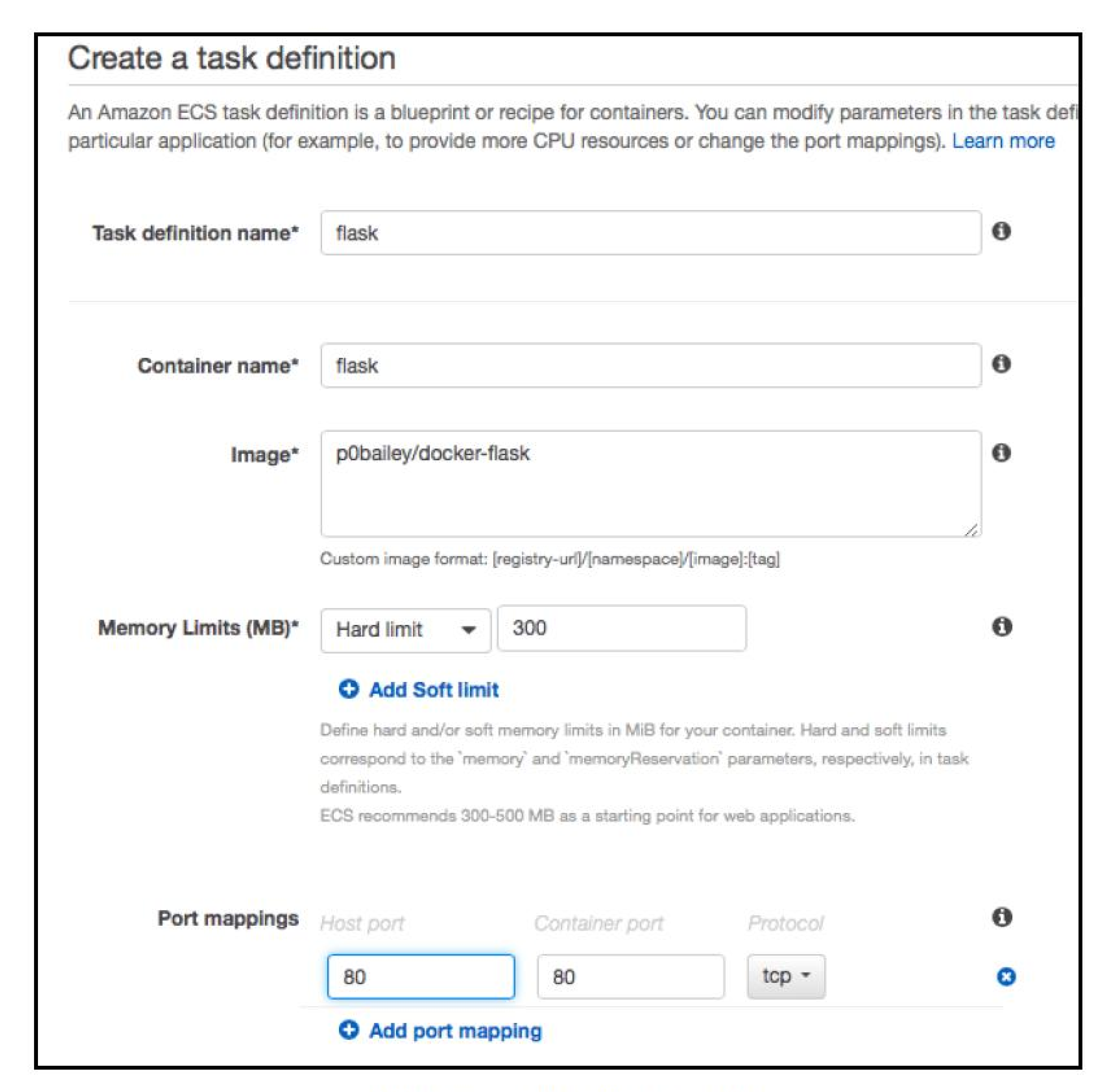


图 11-9　定义任务名和容器

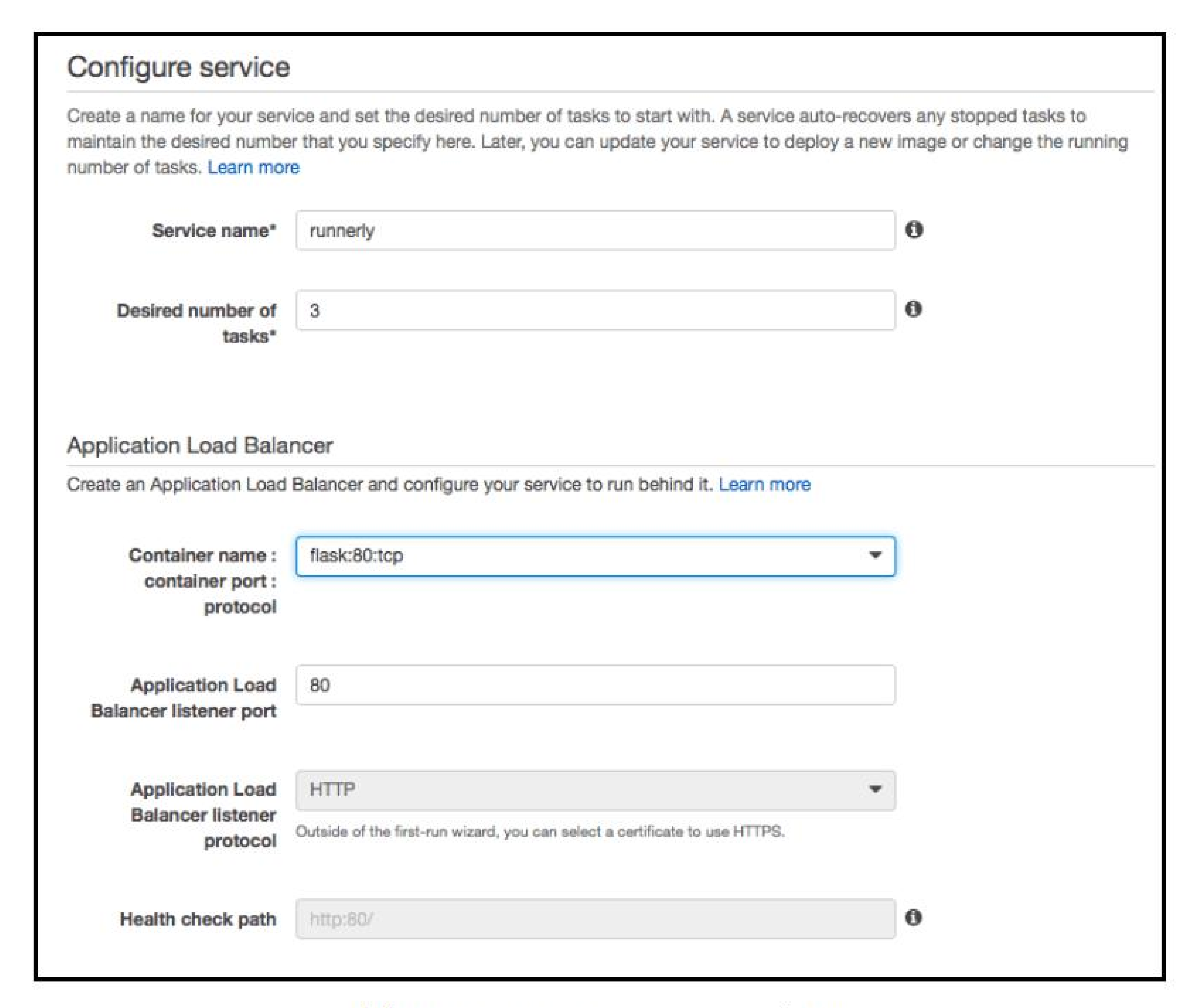


图 11-10　Configure service 窗口

我们给服务添加三个任务，希望在集群中运行三个实例，每个 Docker 容器运行一个。在 Application Load Balancer 区域中，使用之前的容器名和端口。最后，需要配置一个集群，这里需要设置实例类型、一些实例以及将使用的 SSH 密钥对，如图 11-11 所示。

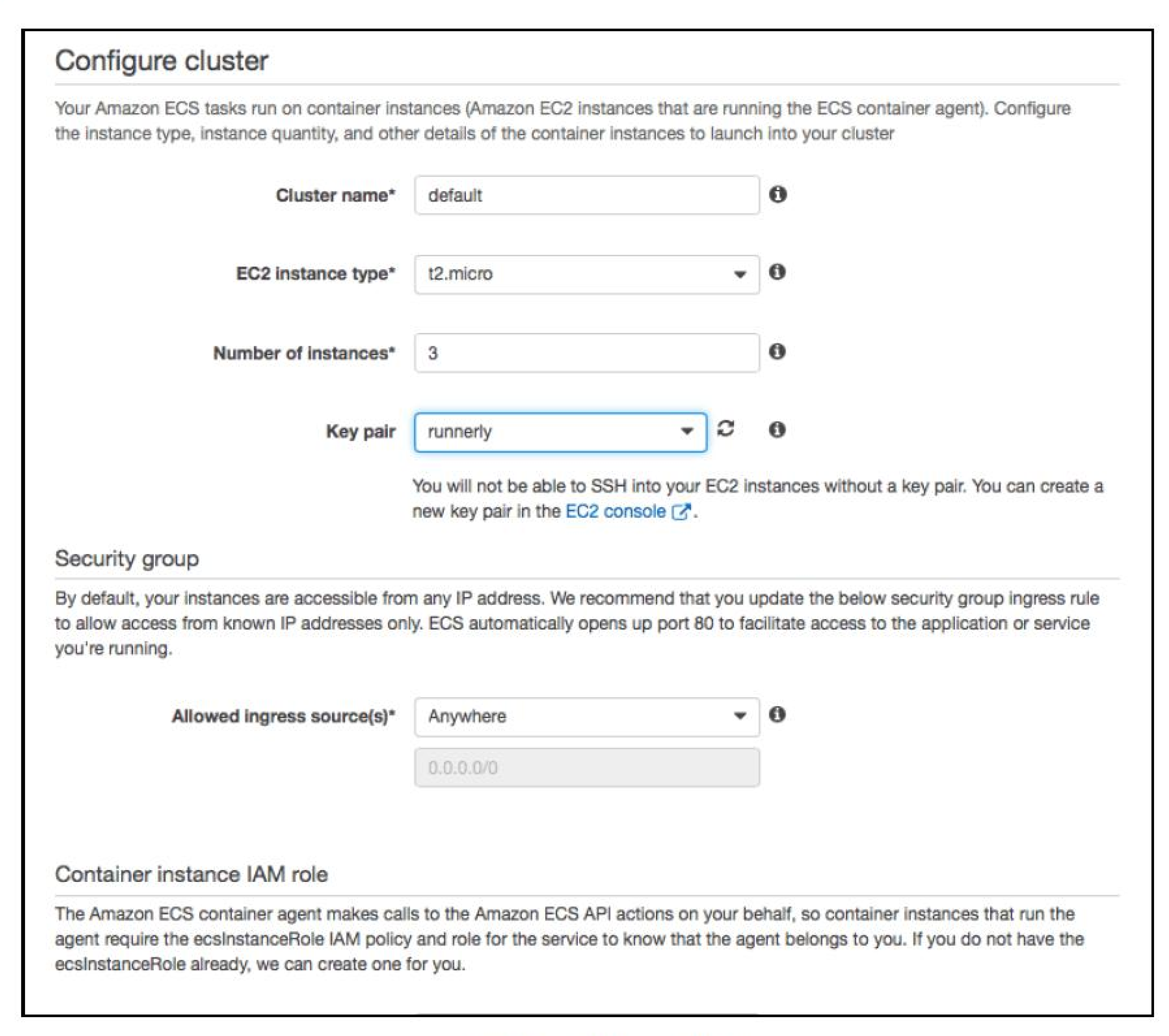


图 11-11　配置一个集群

验证最后一步后，ECS 向导会运行一段时间，来创建所有部件。一旦准备就绪，将看到 view service 按钮，它在所有部件都创建好后变成可点击状态。Service 页面总结部署的所有部件，以及几个用来检查每个服务详情的选项卡，如图 11-12 所示。由 ECS 向导完成的部署可概括如下：

- 创建一个 Task definition 来运行 Docker 容器。
- 添加一个集群，其中有三个 EC2 实例。
- 在集群中添加一个服务，在 EC2 实例中用 Task definition 部署 Docker 容器。
- 通过之前创建的 ELB 实现负载均衡。

如果回到 EC2 控制台，访问左侧的 Load Balancing | Load Balancers 菜单，将看到新创建的 ECS-first-run-alb ELB 已显示在服务 ECS 集群了。

ELB 有一个公开的 DNS 名，这样可通过浏览器访问 Flask 应用。URL 的形式是 http://<ELB name>.<region>.elb.amazonaws.com。

下一节介绍如何将 ELB URL 链接到干净的域名上。

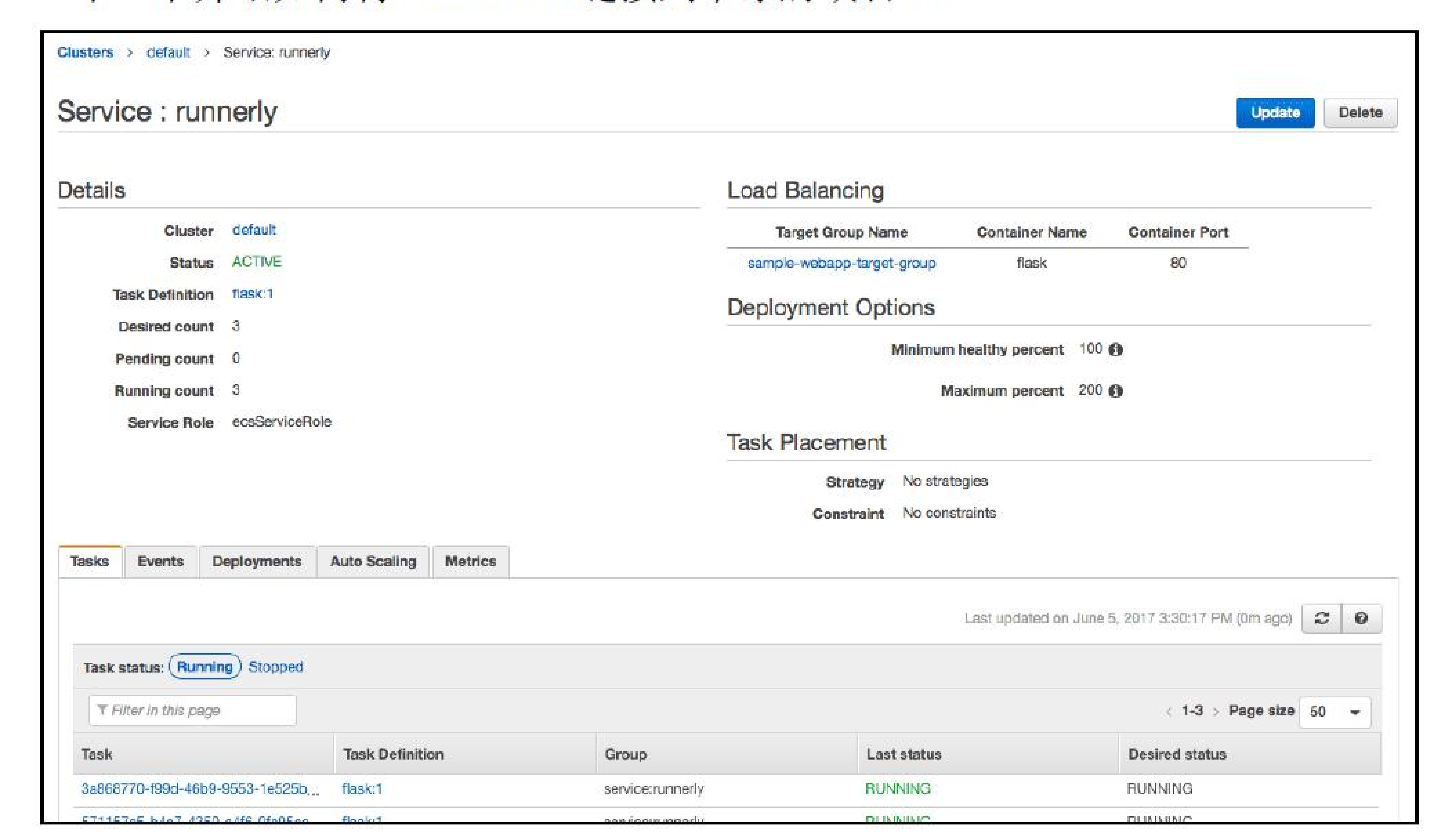


图 11-12　总结部署的所有部件

11.7　Route53

Route53 可用来创建域名的别名。如果在 https://console.aws.amazon.com/route53 上访问 Route53 服务的控制台，单击 hosted zones 菜单，可给域名添加一个新的 Hosted Zone，这是之前创建的 ELB 的别名。

假设你已拥有来自域名注册服务商的域名，那么可简单地将域名重定向到 AWS 的 DNS。单击 Create Hosted Zone，然后添加域名，如图 11-13 所示。

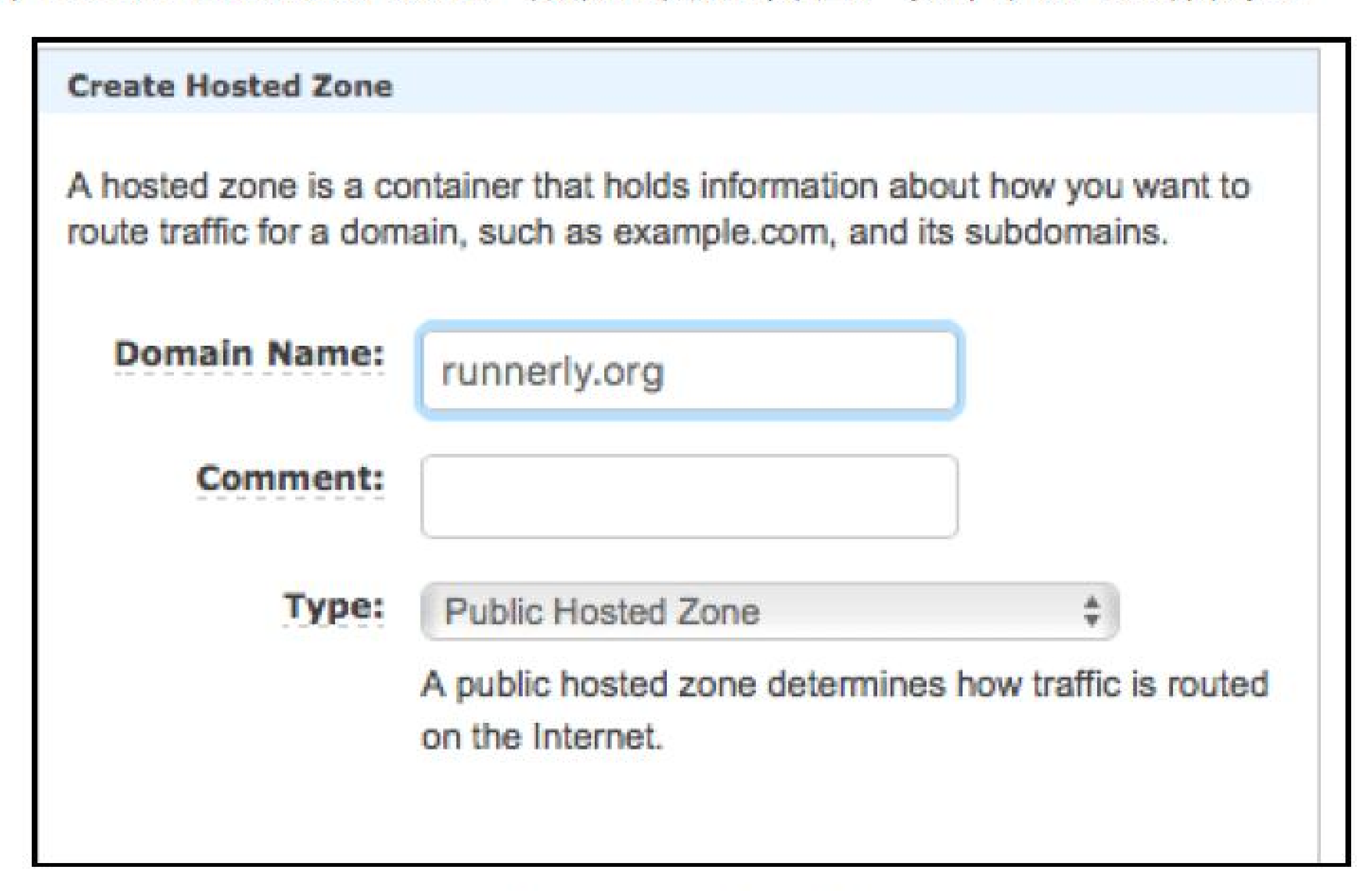


图 11-13　添加域名

创建后，可在 Create Record Set 中选择一个 A 类型记录。该记录必须是别名，并且在 Alias Target 输入框中出现一个包含所有可用项的下拉列表，如图 11-14 所示。

图 11-14　Create Record Set 窗口

前面通过向导创建的 ELB 负载均衡器也应出现在列表中，选择它来链接域名和 ELB。

完成这一步便可将域名链接到已部署的 ECS 群集上。可用子域名增加更多条目，例如每个子域名都对应一个已部署的微服务。

Route53 在全球都有 DNS 服务器，还有其他一些有趣功能，如健康检查等，它可定期连接你的 ELB 和其中的服务。如果连接失败，Route53 自动给 CloudWatch 发送警告。如果有几个 ELB，所有请求会被转发到另一个健康的 ELB 上。

11.8　本章小结

容器化应用正成为部署微服务的标准，云供应商都在追随这种趋势。

Google、Amazon 和其他大型云供应商都支持你部署和管理 Docker 容器集群。所以，如果应用是 Docker 化的，应该能很容易地在这些云上部署。本章介绍如何在 AWS 中部署，AWS 有自己用来管理 Docker 镜像的服务(ECS)，而且与其他 AWS 服务紧密集成。

一旦熟悉所有 AWS 服务，就会发现这是一个非常强大的平台。可发布的不仅是大规模的微服务的应用，也适合部署小应用，相对于运行自己的数据中心，成本较低。

下一章将总结本书，思考构建微服务的艺术。

第 12 章 接下来做什么？

5 年前选择 Python 版本时，要考虑以下两个因素：

- 应用部署到哪种操作系统？
- 应用使用的库是否可用？

这里有一个极端的例子描述了在使用 CentOS 时，操作系统如何影响这个决定。CentOS 与 Red Hat Enterprise Linux(RHEL)非常接近，但移除了商业支持。许多公司从 RHEL 开始培养内部团队，最终迁移到 CentOS 上。有很多使用 CentOS 的理由，例如这个 Linux 发行版非常流行，并构建在一组强大和稳定的管理工具上。

然而，使用CentOS意味着不能在项目上使用最新的Python版本(在系统安装了自定义Python实例的情况除外)。而且，从运维角度看，这通常是一个糟糕的实践，因为没有使用Python社区支持的版本。因此，一些开发者被迫在很长时间内使用Python 2.6，无法使用最新的Python语法和特性。

一些用户认为有些必需的库未迁移到 Python 3，因此仍停留在 Python 2。实际上，这并非事实——如果在 2017 年启动一个新的微服务项目，一切都在 Python 3 中可用。

现在，这两个坚持使用 Python 旧版本的原因已经消失了。你可使用最新的 Python 3，并将应用部署到 Docker 容器中的任何 Linux 发行版上。

如第 10 章所述，Docker 似乎成为容器化应用的最新标准。但其他玩家也可能成为重要替代品，例如 CoreOS 的 rkt(https://coreos.com/rkt/)。无论如何，当所有容器引擎都基于统一标准来描述镜像时，容器技术会成熟起来——这是 OCI(Open Container Initiative)等组织的目标。OCI 由所有大型容器和云服务玩家驱动。

基于上述原因，选择最新 Python 3 和 Docker 来构建微服务是一个安全的赌注。将来，Dockerfile 的语法可能与 OCI 组织构建的语法非常接近。

因此，如果 Python 3.6 及其后续版本具有优良的特性，那么没理由阻止你继续前

进以及在下一个微服务使用它们——如同本书介绍的，在不同微服务上使用不同技术栈或使用不同 Python 版本是没有问题的。

本书选择 Flask，因为这个框架适于构建微服务，并有广泛和成熟的生态系统。但从 Python 3.5 开始，基于 asyncio 库(https://docs.python.org/3/library/asyncio.html)的 Web 框架以及 async 和 await 等新的关键字正成为重要的备选项。

很可能在几年后，其中一个会代替 Flask 成为最受欢迎的框架。这是因为它们在 I/O 密集型微服务上的性能表现非常好，而且开发者们开始接受异步编程方式。

本章将介绍 Python 3.5+中的异步编程是如何工作的，并探索两个可用来构建异步微服务的 Web 框架。

12.1 迭代器和生成器

要理解 Python 中的异步编程如何工作，首先必须理解迭代器(iterator)和生成器(generator)的工作原理，它们是 Python 异步特性的基础。

Python 中的迭代器是实现了迭代器协议(Iterator protocol)的类。这个类必须实现下面两个方法：

- __iter__()：返回真正的迭代器。通常返回 self。
- next()：返回下一个值，直至抛出 StopIteration()异常。

下例将 Fibonacci 序列实现为一个迭代器：

```
class Fibo:
    def __init__(self, max=10):
        self.a, self.b = 0, 1
        self.max = max
        self.count = 0

    def __iter__(self):
        return self

    def next(self):
        try:
            return self.a
        finally:
            if self.count == self.max:
                raise StopIteration()
            self.a, self.b = self.b, self.a + self.b
```

```
            self.count += 1
```

迭代器可直接在循环中使用，如下所示：

```
>>> for number in Fibo(10):
...     print(number)
...
0
1
1
2
3
5
8
13
21
34
```

为让迭代器更具有 Python 特征，Python 添加了生成器，并引入 yield 关键字。当函数使用 yield 而不是 return 时，会被转换成生成器。在执行过程中遇到 yield 关键字时，函数会返回 yield 值并暂停执行。

```
def fibo(max=10):
    a, b = 0, 1
    cpt = 0
    while cpt < max:
        yield a
        a, b = b, a + b
        cpt += 1
```

这样的行为让生成器与其他语言中的协同程序有些相似(但协同程序是双向的)。协同程序能像 yield 那样返回值，还能接收一个值并在下次迭代中使用。

能暂停函数的执行并在两个方向与其通信是异步编程的基础。一旦具备这样的能力，就能使用事件循环，并暂停和恢复函数的执行。

通过 send()方法来扩展 yield 的调用，就可接收调用方传递的值。下面的例子中，terminal()函数模拟一个终端，实现了三个指令：echo、exit 和 eval：

```
def terminal():
    while True:
        msg = yield # msg gets the value sent via a send() call
```

```
        if msg == 'exit':
            print("Bye!")
            break
        elif msg.startswith('echo'):
            print(msg.split('echo ', 1)[1])
        elif msg.startswith('eval'):
            print(eval(msg.split('eval', 1)[1]))
```

实例化这个生成器后，就能用 send()方法来接收数据：

```
>>> t = terminal()
>>> t.next() # call to initialise the generator - similar to send(None)

>>> t.send("echo hey")
hey

>>> t.send("eval 1+1")
2

>>> t.send("exit")
Bye!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

注意，在最新的 Python 3 中，这段代码中的 t.next()需要改成 next(t)或 t.__next__()。

有了上面添加的代码，Python 生成器变得类似于协同程序。

对 yield 的另一个扩展是 yield from，它允许链式调用另一个生成器。

考虑下面的例子，一个生成器使用其他两个生成器来生成(yield)值：

```
def gen1():
    for i in [1, 2, 3]:
        yield i

def gen2():
    for i in 'abc':
        yield i

def gen():
    for val in gen1():
```

```
        yield val
    for val in gen2():
        yield val
```

可使用一个 yield from 调用来替换上面代码 gen()里的两个 for 循环：

```
def gen():
    yield from gen1()
    yield from gen2()
```

下面是调用 gen()方法的例子，它在每个子生成器调用完毕后返回：

```
>>> list(gen())
[1, 2, 3, 'a', 'b', 'c']
```

调用其他多个协同程序并等待它们执行完毕是异步编程的一个流行模式。它允许开发者将逻辑拆分成较小的函数并依次组装起来。每个 yield 调用都是函数暂停执行并让其他函数接管的机会。

有了这些特性，Python 在支持原生异步编程上更进一步。过去，使用迭代器和生成器构建代码块来创建原生的协同程序。

12.2　协同程序

为更直观地进行异步编程，Python 3.5 引入 await 和 async 关键字以及 coroutine 类型。await 调用几乎等价于 yield from，目标是从一个协同程序中调用另一个协同程序。

await 与 yield from 的区别在于，不能使用 await 调用一个生成器。

async 关键字能标记函数、for 循环或 with 块，使其成为一个原生协同程序。如果使用这个函数，将得到一个 coroutine 类型的对象，而不是生成器。

添加到 Python 中的原生 coroutine 类型就像一个完全对称的生成器，但所有往返的调用都被代理到一个事件循环中。事件循环负责协调程序的执行。

下面使用 asyncio 库来运行 main()函数，它并行调用多个协同程序：

```
import asyncio

async def compute():
    for i in range(5):
        print('compute %d' % i)
        await asyncio.sleep(.1)
```

```
async def compute2():
    for i in range(5):
        print('compute2 %d' % i)
        await asyncio.sleep(.2)

async def main():
    await asyncio.gather(compute(), compute2())

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
loop.close()
```

值得注意的是，除了使用 async 和 await 关键字，它与普通顺序执行的 Python 代码十分相似，这意味着可读性很高。与使用线程不同，协同程序通过让出控制权(而不是中断调用)来工作，所以执行顺序是确定的，每次运行发生的事件都相同。

注意 asyncio.sleep()是一个协同程序，所以使用 await 关键字来调用它。

如果运行这个程序，将得到以下输出：

```
$ python async.py
compute 0
compute2 0
compute 1
compute2 1
compute 2
compute 3
compute2 2
compute 4
compute2 3
compute2 4
```

下一节将介绍 aysncio 库。

12.3 asyncio 库

asyncio(https://docs.python.org/3/library/asyncio.html) 库最初称为 Tulip，是 Guido(Python 的作者)的一个实验。asyncio 提供用来构建基于事件循环的异步程序的所有基础设施。

这个库的出现早于 Python 语言的 async 和 await 以及协同程序。

asyncio 库的灵感源于 Twisted，提供若干模仿 Twisted 传输和协议的类。为构建基于这些类的应用，需要结合使用传输类(例如 TCP)和协议类(例如 HTTP)，然后使用回调方式将不同部分编排起来。

但引入原生协同程序后，回调风格的编程方式就不再吸引人了。通过 await 调用来编排执行顺序的可读性更高。虽然可通过 asyncio 协议和传输类来使用协同程序，但 asyncio 最初并非为它设计，而且需要一些额外工作才能使用。

然而，最主要的特性是事件循环 API，以及用来调度协同程序执行的所有函数。事件循环通过操作系统的 I/O 轮询器(如 devpoll、epoll 和 kqueue)将给定 I/O 事件对应的执行函数注册到系统中。

例如，事件循环能在 socket 上等待数据可用后，触发一个处理数据的函数。但这种模式可推广到任何事件上。例如，假定协同程序 A 等待协同程序 B 完成后才执行，那么使用 asyncio 设定一个 I/O 事件，它在协同程序 B 结束时触发，并会让协同程序 A 等其发生后继续执行。

这样，如果程序被拆分成许多相互依赖的协同程序，它们就会交错执行。这个模式的美妙之处在于，一个单线程的应用可并发运行上千个协同程序，而这些协同程序不必是线程安全的，也不需要为线程安全而引入相应的复杂性。

构建异步的微服务时，典型模式如下：

```
async def my_view(request):
    query = await process_request(request)
    data = await some_database.query(query)
    response = await build_response(data)
    return response
```

这个协同程序处理每个传入的请求，事件循环运行后，能在等待每一步完成时，接收上百个新请求。

如果这个服务使用 Flask 构建，并以单线程方式运行，那么每个新请求都只有等待上一个请求完成才能被 Flask 应用处理。数百个并发请求指向该服务，服务器会立即返回超时。

在两种情况下，每个请求的执行事件都相同。但并发处理大量请求并交错执行的能力，使得异步应用更适合 I/O 密集型微服务。应用可在等待数据库调用返回时，使用 CPU 做很多事情。

如果一些服务包含 CPU 密集型任务，asyncio 提供一个函数，能在事件循环之外通过单独线程或进程执行相应的代码。

下面介绍两个基于 asyncio 并用来构建微服务的框架。

12.4 aiohttp 框架

aiohttp(http://aiohttp.readthedocs.io/)是基于 asyncio 的流行框架，它在 asyncio 的早期就已出现。

与 Flask 类似，它提供一个请求对象和一个路由，将查询转发到对应的处理函数上。

asyncio 库的事件循环被封装在 Application 对象中，它处理大部分编排工作。作为微服务的开发者，可像使用 Flask 那样专注于构建视图上。

下面的例子中，当访问应用的/api 时，协同程序 api()返回 JSON 响应：

```
from aiohttp import web

async def api(request):
    return web.json_response({'some': 'data'})
app = web.Application()
app.router.add_get('/api', api)
web.run_app(app)
```

aiohttp 框架有一个内置的 Web 服务器，可使用 run_app()方法来运行这个脚本。总的来说，如果习惯了 Flask，那么最大的不同在于此处没有使用装饰器将请求路由到视图上。

这个框架提供若干个能在 Flask 中找到的帮助程序和一些独创特性，例如中间件。使用中间件可通过注册协同程序执行特定任务(例如自定义的错误处理)。

12.5 Sanic

Sanic(http://sanic.readthedocs.io/)是另一个有趣项目，它使用协同程序并尝试提供类似于 Flask 的体验。

Sanic 使用 uvloop(https://github.com/MagicStack/uvloop)作为事件循环。uvloop 是一个使用 libuv，并用 Cython 实现了 asyncio 循环的协议。在大多数微服务中，这个区别微不足道，但透明地切换到一个特定事件循环并获得一定的速度提升，也是不错的。

如果使用 Sanic 重写前面的例子，它与 Flask 非常相似：

```
from sanic import Sanic, response

app = Sanic(__name__)

@app.route("/api")
async def api(request):
    return response.json({'some': 'data'})

app.run()
```

不用说，整个框架受到 Flask 的启发。你会在 Sanic 中找到几乎所有可让 Flask 成功的特性，例如 Blueprint。

Sanic 也有其独创特性，例如在类(HTTPMethodView)中编写视图的能力。这个类代表一个端点，每个动词(GET、POST 和 PATCH 等)对应于它的一个方法。

这个框架也提供中间件来修改请求或响应。

在下例所示，如果视图函数返回字典，框架会自动将其转换成 JSON：

```
from sanic import Sanic
from sanic.response import json

app = Sanic(__name__)

@app.middleware('response')
async def convert(request, response):
    if isinstance(response, dict):
        return json(response)
    return response

@app.route("/api")
async def api(request):
    return {'some': 'data'}

app.run()
```

如果微服务仅返回 JSON 映射，那么这个小中间件函数就能简化视图代码。

12.6 异步和同步

切换到异步模型意味着需要在所有地方使用异步代码。

例如，如果微服务使用一个请求库，但不是异步的，那么每个查询 HTTP 端点的请求都会阻塞事件循环，此时并不能从异步中获益。

将一个现有项目改为异步模式并不容易，因为这需要彻底修改它的设计。大多数想支持异步调用的项目都从头设计一切。



好消息是，有越来越多的可用异步库可用来构建微服务。在 PyPI 上，可搜索 aio 或 asyncio。这个 wiki 页面(https://github.com/python/asyncio/wiki/ThirdParty)也是一个值得查看的好地方。

下面列出一些与构建微服务相关的库：

- **aiohttp.Client**：可用来替换 requests 包。
- **aiopg**：构建在 Psycopg 上的 PostgreSQL 驱动。
- **aiobotocore**：AWS 客户端——今后可能被合并到 boto3 官方项目中。
- **aioredis**：Redis 客户端。
- **aiomysql**：MySQL 客户端，基于 PyMySQL 构建。

若找不到某些库的替代品，asyncio提供一个执行器(executor)，可用来在独立线程或进程中执行阻塞代码。这个函数是一个协同程序，底层使用concurrent模块中的ThreadPoolExecutor或ProcessPoolExecutor类。

下例通过线程池来使用 requests 库：

```
import asyncio
from concurrent.futures import ThreadPoolExecutor
import requests

# blocking code
def fetch(url):
    return requests.get(url).text

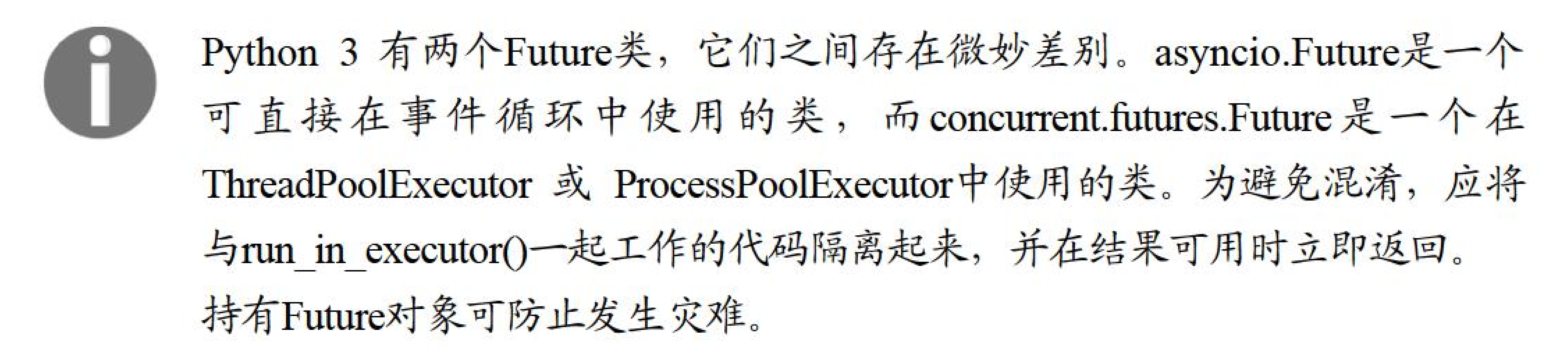
URLS = ['http://ziade.org', 'http://python.org', 'http://mozilla.org']

# coroutine
async def example(loop):
    executor = ThreadPoolExecutor(max_workers=3)
    tasks = []
```

```
        for url in URLS:
            tasks.append(loop.run_in_executor(executor, fetch, url))
        completed, pending = await asyncio.wait(tasks)
        for task in completed:
            print(task.result())

loop = asyncio.get_event_loop()
loop.run_until_complete(example(loop))
loop.close()
```

每次调用 run_in_executor()都会返回一个 Future 对象，它用来在异步程序中设置同步点。Future 对象会监视执行状态，并提供一个方法在结果可用时获取结果。



Python 3 有两个Future类，它们之间存在微妙差别。asyncio.Future是一个可直接在事件循环中使用的类，而concurrent.futures.Future是一个在ThreadPoolExecutor 或 ProcessPoolExecutor中使用的类。为避免混淆，应将与run_in_executor()一起工作的代码隔离起来，并在结果可用时立即返回。持有Future对象可防止发生灾难。

asyncio.wait()函数能等待所有 Future 对象的完成，因此这里的 example()函数会在所有 Future 返回前阻塞。wait()函数可接收超时时间，所以返回一个元组，包含已完成的 Future 和仍在执行的 Future。如果不指定超时时间，将无限期地等待(除非在 socket 库设置了全局超时时间)。

可用进程来替代线程，但这种情况下，所有进入和离开阻塞函数的数据都必须是可序列化的。最好完全避免阻塞代码，尤其在代码是 I/O 密集型时。

这就是说，如果你有一个 CPU 密集型函数，那么在独立进程运行它是值得的，因为这样做能利用所有可用的 CPU 核心，并加速微服务。

12.7　本章小结

本章讲述如何在 Python 中使用异步编程来编写微服务。尽管 Flask 是一个很棒的框架，但异步编程可能成为使用 Python 编写 I/O 密集型微服务的下一场革命。

基于 Python 3.5 及更高版本的异步框架和库越来越多，这使得这种方法很有吸引力。

选择一个微服务，然后从 Flask 切换到其中一个框架，可能是在有限风险下尝试异步模式的好方法。